VZ200/300

ASSEMBLY LANGUAGE PROGRAMMING MANUAL FOR BEGINNERS

Steve Olney

VZ200/300

ASSEMBLY LANGUAGE PROGRAMMING MANUAL FOR BEGINNERS

by Steve Olney

FIRST EDITION 1987

National Library of Australia

ISBN 0 9587808 0 3

Published by:-

S.R. Olney
P.O. Box 125
North Richmond 2754.

Printed in Australia by:-

G. Moorcroft
Bowen Mountain Road
Grose Vale 2753.

VZ200/300

ASSEMBLY LANGUAGE PROGRAMMING MANUAL FOR BEGINNERS

by Steve Olney

PREFACE

This manual is a practical beginners guide to machine code programming using the VZ200/300. It is meant as a simple introduction to machine code programming for those who want to further explore the capabilities of their machine beyond the boundaries imposed by Basic.

The manual explains what machine code is, how to write simple machine code programs, and how to pass these programs to your computer. It also contains some information about VZ200/300 hardware to allow you to more fully utilise the power of machine code programs.

The transition from Basic programming to machine code programming is a large and difficult step to take. Machine code is complicated and is an order more difficult than Basic programming with many rules to follow. Fortunately though, the rewards for the machine code programmer are high. For high speed operation in graphics, or special cassette reading routines, machine code is a must.

There are many general 'how to' texts available on machine code or Assembly Language, but the problem is the vast majority are either not specific enough for the VZ, or pitched at too high a level for the beginner. This manual does not attempt to repeat the more detailed, general information available in such texts, but is meant to be a primer for the VZ owner just starting to learn, allowing them to take the first steps on that long, but fascinating road which is machine code programming.

As you work through this manual it is assumed that you have beside you the Basic Reference Manual supplied with your VZ, as well as one of the many 'How To Program The Z-80' books available. If you can also get hold of the VZ Technical Reference Manual this will be of great assistance.

I hope the information contained in these pages is of use and interest to you and provides encouragement to continue your studies in this area, thereby providing many hours of enjoyment as it has done for me.

Steve Olney

CONTENTS

CONTENTS - continued

CONTENTS - continued

CONTENTS - continued

APPENDICES

NOTES

CHAPTER 1

INTRODUCTION TO MACHINE CODE

Machine code is the code that the computer uses directly and takes less room than the equivalent program in Basic and usually runs much faster. On the minus side it is much more difficult to write and also read, even one that you have written yourself just recently.

The instructions given in this book give you a stage by stage introduction to the basics of machine code and show you how to write simple programs yourself. Examples will be given of adding two numbers together and displaying a message on the screen. Also, details on how to load and run machine code from Basic programs will be given.

On a slightly more advanced note, the use of an Editor Assembler to simplify the writing of machine code is explained.

The main reason machine code is hard to read, is that it is not written in an English-like fashion as Basic is, but when examined in memory it appears as a list of numbers and letters. This makes finding 'bugs' very difficult. Also because the effects of bugs are usually catastrophic. Generally, this means that the damage has been done before you have time to see where the problem lies.

When you are running a machine code program you are working outside the normal operation of Basic. Therefore, when an error occurs you aren't provided with a helpful message which gives a hint as to where or what the error is. Instead the most common result is a very convincing play-dead-possum act, or sometimes a spectacular video display.

Either effect is far from the desired result, and it is important to be very careful and meticulous when writing your programs to ensure your code contains as few errors as possible. The time taken to check your work two or three times is always a good investment towards speedy completion of your programs.

1.Ø1 WHAT IS MACHINE CODE ?

Machine code is the code that the computer executes directly, i.e., the code which the computer takes from memory and reads as actual instructions to carry out certain operations directly.

You may be surprised to learn that the programs you have written in Basic are not really computer instructions in the direct sense, but rather a list of data items.

There is a large block of machine code called a Basic Interpreter already built into your VZ and a part of this machine code translates your Basic listing as a sequence of calls to other sections of the same inbuilt machine code in order to carry out the tasks you have specified with your Basic listing.

Each part of a machine code program is represented by a binary number. Sometimes the binary number represents an instruction to the computer and sometimes it represents a number directly.

Which case it represents depends on the order and context of the binary number within the machine code program. A binary number is a form of representation of numerical quantities and data directly allied to what happens at the hardware level of operation of the computer.

1.02 DIGITAL VOLTAGE LEVELS

In a digital system like your VZ computer, data is moved around as a pattern of two voltage levels. In your VZ, as is the case in practically all microprocessor controlled computers, these two levels are +5V and 0V. If you attach an oscilloscope to the digital part of your computer circuits you will observe these two levels, some static, and some changing at different rates.

To relate these purely physical voltage levels to the representation of data these levels are called by names other than +5V and 0V, i.e. 'high' and 'low', or 'logic 1' and 'logic 0' respectively.

1.03 TWO LEVELS - BINARY NUMBERS

If we are using these binary levels in writing down machine code data or using them in arithmetic operations, then we use the symbols '1' and '0' respectively. We can write any number in terms of binary, for example, the number '25' in binary is '00011001'.

We could enter the number 25 into a digital system by setting up the position of eight switches, reading left to right, the first three off, the next two on, the next two off and the last one on.

1.Ø4 BINARY DATA IN THE COMPUTER

In the computer the switches are replaced with a set of eight data lines running around inside, connecting one chip to another. This is called the data bus.

Along this data bus information can be moved back and forth between the various sections of the computer which needs to have access to the data. The grouping of these eight data lines each carrying a 'bit' of information (either '1' or 'Ø') is called a binary data 'byte'.

Each binary byte carries the code for one piece of information or data for use by the computer. These binary bytes are the codes which the computer uses directly to carry out the tasks set to it. A sequence of these binary bytes is called a machine code program.

Later on we will look at binary more closely, but while we are in the early stages we will be using a numbering system which is closely related to both binary and our ordinary everyday decimal counting system. This numbering system is called hexadecimal, which Chapter 3 deals with in more detail.

1.Ø5 MACHINE CODE IS COMPLEX

In a Basic program the task of adding two numbers together is written out in an English-like form, i.e.,

```
PRINT 3 + 4
```

In machine code however, each step of the operation must be specified, as the power of each instruction to the computer is very limited, i.e., moving data, or adding and subtracting two numbers stored in specified areas of the computer. Also the size of the numbers that can be handled in one operation is restricted.

In Basic programs all these details are taken care of for us by the Basic Interpreter, but in machine code we are strictly on our own.

To further emphasise the difference, consider the task of clearing the video screen. In Basic this is done by simply typing 'CLS', while in machine code programming you must specify where in the computer's memory the screen area begins, how many locations need to be cleared, and what byte code must be entered into the screen video area to produce a blank screen.

Many times you will find yourself thinking that this or that task could be much more easily done in Basic, but persevere, as the same can be said about learning any new human language, say French.

In the early stages, of course you can use the old familiar English language better. But finally, the new language opens up areas which are not accessible by the old. So it is with learning machine code.

1.Ø6 INTRODUCING HEXADECIMAL

When writing machine code you could write down all the instructions in binary numbers, but this would be very slow and confusing and the last thing we need is confusion. Instead we can use a shorter, more compact representation called hexadecimal.

Once you are familiar with hex (short for hexadecimal) you will find it easier to use than binary. In any case, it is very easy to convert hex to binary if needs be. The hexadecimal numbering system uses sixteen different symbols, taking Ø-9 from the decimal numbering system and adding another six from the alphabet, A-F. That's six letters plus ten numbers, hence **HEXaDECIMAL.**

1.Ø7 HEXADECIMAL CODES

The following could be a hex representation of a machine code program:-

```
CD 7A Ø1
3E 9A
21 FF FF
```

Each line starts with an instruction, the next byte(s), if any, containing data. With the Z8Ø sometimes the second byte is also part of an instruction code. This is because the Z8Ø has more instructions than can be uniquely specified by just one byte. Each pair of symbols is the hex equivalent of the binary code that would appear, in turn, on the data lines, if the computer was running the above program.

1.Ø8 MNEMONICS AND ASSEMBLY LANGUAGE

We can also write down the instructions in a kind of shorthand form, called 'assembly language', consisting of 'mnemonics' which indicate what the instruction does.

For example, the above short program could be written in assembly language form thus:-

```
CALL      Ø17AH
LD        A,9AH
LD        HL,ØFFFFH
```

The second line 'LD A,9AH' can be read, "load register A with the hex number 9A". This makes the program much easier to read and debug.

1.Ø9 WHAT IS AN EDITOR ASSEMBLER ?

Of course, the mnemonics cannot be read by the computer directly, so we use a translater program called an Editor Assembler to convert the assembly language source into the correct sequence of binary codes in memory. This translater program, unlike the Basic Interpreter, does not come inbuilt to the VZ, but must be loaded from tape or disc.

The editor part of the Editor program allows us to build up our assembly source listing, while the assembler part does the translation of the mnemonics into machine code. We can get a printout of the machine code in hex, and also generate a tape which contains the machine code program stored in binary form ready to be loaded into memory and run or executed.

1.1Ø THE BASIC LOADER

In the early stages we will use the method of writing the program in mnemonic form, then once we are certain that the logic of the program is correct, translate the mnemonics into their hex equivalents. You can then use a Basic 'loader' program to take the hex numbers in DATA statements and convert them into decimal suitable for loading into memory from Basic.

Alternatively, you can convert the hex numbers yourself by hand (or calculator conversion) to decimal and enter the decimal equivalents into the DATA statements, ready to be loaded into memory by Basic. The latter method has the advantage of being faster when loading the code, but is more prone to errors than the first method.

-ooOoo-

NOTES

CHAPTER 2

PARTS OF YOUR VZ COMPUTER

As mentioned before, when you are programming in machine code you must specify every step in the operation of the program. You must tell the computer (via the program) where the video screen is located, where to store or retrieve data and so on.

When you are programming in Basic all this information is already there and is used by the interpreter to carry out all these operations transparently. That is, the programmer does not need to know where everything is stored in the computer's hardware in order to program.

However, to do any useful work in machine code, you must know something about the hardware side of the computer. The following is intended to give a brief idea of the operation of some of the major parts inside your VZ.

2.01 PLASTIC CHIPS

If you were to open up the case of your VZ you would find a number of black rectangular plastic-looking components with pins on each side. These are integrated circuits (IC for short, or more commonly - chips). They are called integrated circuits because a number of different circuit components have been etched onto one or two silicon slices inside the plastic package.

It is this integration of complex circuits onto a small area which has allowed the production of sophisticated devices which not long ago were affordable only by universities or other government institutions, and led to the development of the personal computer.

2.02 DATA AND ADDRESS BUSSES

Data flows between the chips in the form of eight-bit bytes via the data bus, while the locations from where, and, to where the data flows is specified by the binary number on a bus sixteen bits wide, called the 'address' bus. The control of which chip is receiving and which chip is transmitting the data is specified by several lines called collectively the 'control' bus.

2.03 MEMORY CHIPS

ROM stands for 'read only memory', and is the name for the chips where permanent information is stored in the computer. These chips contain, in the case of the VZ, the machine code instructions for the operation of the Basic Interpreter.

Every time you switch on the VZ, you find that the Basic Interpreter starts running by itself. This is because the information stored in the ROM is retained even when the power is switched off. The information has been burnt into the ROM before it was soldered into the circuit and cannot be overwritten by the computer, hence the name 'read-only'.

What this means as far as our machine code programming is concerned, is that we cannot use the space occupied by ROM to store our programs, simply because we cannot change the information stored at these locations.

RAM stands for 'random access memory' and this is where you will store your programs which you will load into the computer. Not only can the information be read out of these chips, but also new information can be stored in them. When you load different Basic programs, either by typing them in or by loading from tape and disc, this is where the information is stored.

2.04 THE MICROPROCESSOR CHIP

The microprocessor chip holds the CPU, or the central processing unit, which co-ordinates all the processing of the data within the computer. It is the brains of the computer, and performs the addition, subtraction, comparison and movement operations on the data.

2.05 CLOCK TIMING

The clock is the regulator of the events that occur inside the computer. It synchronises the transfer of data by timing via clock pulses and ensures that all parts of the computer have sufficient time to respond to the information supplied to it.

If you view the clock signal on an oscilloscope, you will find that it is just a square signal switching between logic 1 and logic 0 levels. It is called a clock because of its timing and synchronising role in the computer, and has no direct connection with hours, minutes and seconds.

2.Ø6 ADDRESS SPACE

The CPU inside the computer is the chip which produces the address information which is output on the address bus. Different addresses on the address bus specify one single location in the computers memory. The CPU can put out an address number on the address bus for 65,536 different locations.

This address space can be filled with memory chips, either ROM or RAM, and in the case of the VZ, special circuits which allow communication with external inputs, like the keyboard.

When we are programming in machine code we need a map to show us where ROM and RAM and other circuits are located in the address range, so that we don't accidently try to occupy locations which are reserved for other purposes.

Also, when Basic is operating, it needs room to store variables from the program and temporary pointers and data for its own use. If we inadvertently overwrite these areas, Basic will cease to operate correctly.

In Chapter 7, which gives more information about the partitioning of the VZ memory, we will look in more detail at how the VZ memory is utilised.

2.Ø7 WORK SPACE

The CPU makes use of some RAM space for work areas. It needs space for buffers which are temporary stores for incoming data such as input from a keyboard or cassette interface or disc drive.

When Basic is being used the CPU needs space for the Basic RETURN stack, for storing the return addresses for use by the GOSUB command.

A system variables scratchpad is needed to store information about the current status of the program being executed, for example, the last DATA item read by the READ command.

Additionally, the CPU needs its own machine code return stack used to store return addresses for machine code CALLs from within the Interpreter machine code itself.

2.Ø8 PAGES IN MEMORY

Memory addresses can be conveniently thought of being divided into pages, with each page containing 256 locations. The Z8Ø CPU can address 256 pages giving 256 x 256 = 65,536 locations altogether.

Thinking of the address range as divided up in this way is useful because address locations are stored in two bytes in a machine code program, the first byte specifying the position in page, while the second specifies the page number. Page and location numbers begin from 0 and end with 255. That is, the hundredth byte would appear in location 99 on page 0.

2.09 TRAFFIC ALONG THE DATA BUS

Locations are specified by a unique address, with each location capable of storing eight bits of one data byte. All eight bits are stored and retrieved in byte-size chunks (oh dear !) by the CPU outputting the correct address on the address bus, and then, after waiting a short time for the memory chips to respond, reading or writing the byte on the data bus.

The major part of the work done by the CPU is fetching data from memory, acting on that data, and sometimes, returning the data to memory again. Also the CPU controls the transfer of data between chips, say from a buffer area somewhere in a program to a section of RAM which is used for the video display.

Before we talk about actually writing machine code and placing it into memory, we need to understand how to use hex notation so that we can more easily handle the binary data which makes up the machine code program. The next chapter deals with machine code and the use of hex numbers.

-ooOoo-

CHAPTER 3

MACHINE CODE - AND HEXADECIMAL NUMBERS

The terms 'machine code' and 'assembly language' seem to be used interchangeably. For example, you often hear people talking about writing a program in machine code, and in the same time talking about assembly language. In fact, the end product of both these processes is the same; a series of instructions on which the microprocessor chip inside the computer can act upon directly. That series of coded instructions is called a 'machine code' program.

There are 256 unique arrangements of bits in an eight-bit byte, therefore, there are 256 unique pieces of information that can be represented. Each memory location in your VZ can contain one eight-bit byte, correspondingly, there are eight data lines running around inside forming the data bus which carries the level of each of the eight bits.

It is important to realise that the pattern of 1's and 0's produced by the eight bits is just a code for a piece of information. The assignment of the codes may or may not follow some kind of mathematical order. Also, under different circumstances the same code can represent entirely different information.

For example, the eight-bit byte binary 01000001 can represent the number 65 if you are doing arithmetic operations, while to a video character generator chip this same pattern would represent the letter 'A'.

Our task, in order to control the operation of the CPU, is to place in this memory a correct sequence of bytes which the CPU will interpret as a sequence of instructions.

3.01 HAND ASSEMBLING

To place the bytes of a machine code program in memory we can look them up in a table and load them in one by one. Here we are working directly with the byte codes and so is often called 'hand assembling'. To simplify writing these codes down we use various shorthand representations using different numbering systems, however, we still have to write down the bytes one by one.

In addition, we need to calculate the size of relative jumps in the program.

All in all, it is a tedious process, very prone to errors, and it would be much easier if we could write the instructions down as short words which sound like the instruction they represent - a 'MNEMONIC'.

If we had a special program which could read these mnemonics and arrange or 'assemble' them into the correct sequence of binary bytes, then our task would be easier and much less prone to errors.

A sequential list of mnemonics is called an 'Assembly Language Source Listing', and the program used to write and assemble the source listing is called an 'Editor Assembler'.

Below is an example of machine code written in a numbering system called hex, short for hexadecimal. Beside it is the equivalent assembly language source listing, and finally the binary bit patterns they represent.

MACHINE CODE (Hexadecimal)	ASSEMBLY LANGUAGE (Source Listing)	BINARY CODE (Pattern of Bits)
3E 02	LD A,02H	00111110 00000010
C6 04	ADD A,04H	11000110 00000100
32 57 7F	LD (7F57H),A	00110010 01010111 01111111

3.02 HEXADECIMAL NUMBERS

From the above example we can see that to represent a small quantity (3E hex or 62 decimal) in binary, we require a relatively large number of symbols (ones and zeroes).

This makes manipulating binary notation cumbersome. Several other numbering systems have been devised to make the use of binary numbers easier.

The most often used system is hexadecimal. As the name implies, there are sixteen different symbols available for counting. This means that we run out of symbols in each column at slower rate than for both binary and decimal systems. This makes hex numbers relatively compact.

On the next page is a table showing equivalent numbers for the three systems.

TABLE 3.1

DECIMAL	BINARY	HEXADECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

In the familiar decimal system when we have to write down a number larger than nine, we create a new column to the left to indicate the overflow. Each time there is an overflow, the new column is incremented by one.

There comes a time when this new column overflows, so we create a third column to the left of the first two, and so on, creating as many columns as is necessary to represent the quantity we wish to write down.

Although each column contains only the digits from 0 to 9, the actual value those digits contribute to the total of the number depends on which column they occupy. The effective value of each digit in each column is found by multiplying the digit value (0-9) by the column weighting.

The column weights for the decimal system are, starting from right to left, 1, 10, 100, 1000 and so on. That is, 1, 10, 10x10, 10x10x10.

For the hexadecimal system there are sixteen different symbols, therefore the column weights are, 1, 16, 16x16, 16x16x16 and so on.

Thus the hex number 3BDH is evaluated into the decimal system as;

DIGIT VALUE :	3 (3)	11 (0BH)	13 (0DH)
COLUMN WEIGHT:	16x16 (256)	16	1
COLUMN VALUE :	3x256 (768)	11x16 (176)	13x1 (13)
TOTAL VALUE :	768 + 176 + 13 = 957 DECIMAL		

3.Ø3 BINARY TO HEXADECIMAL

The usefulness of the hexadecimal numbering system can be seen when a conversion from binary to hex is required. Because there is a direct relationship between each hex symbol (or digit) and four bits of a binary number, the conversion process can be performed by just looking up the table given previously and writing the new number.

Using the binary number (111110), we start from the right-hand side of the binary number and select out the four right-most bits (11 / 1110). If we look up Table 3.1 we see that this bit pattern corresponds to the hex digit ØEH. Then we select out the next four bits of the binary number (0011 / 1110).

Notice how I have filled out the left-most three bits with zeroes in order to bring up the total number of bits to four. This corresponds to the hex digit 3H.

Hence: 111110 binary = 3EH hex

Note that when writing hex numbers you must follow the number with the letter 'H' to distinguish it from 33 decimal. This is the convention used by most Editor Assemblers. Incidently,if the hexadecimal number begins with a letter (A-F) then it must be preceded with a 'Ø' to enable the Editor Assembler to distinguish the number from a label, e.g. ØB3H.

3.Ø4 DECIMAL TO HEXADECIMAL

To convert a decimal number, say 1998, to a hex number, first divide the number by 4Ø96 (16x16x16x16) to find how many lots of 4Ø96s there are in the number.

Write down the integer part in hex (Ø in this case). Then divide the remainder by 256 (16x16x16) and write the integer part again in hex (7).

Finally, divide the remainder by 16 and write down the integer part in hex once more (12 decimal or ØCH in hex) and then the remainder of decimal 14 or ØEH. Therefore 1998 decimal = Ø7CEH.

Listings 3.1 and 3.2 are two Basic number conversion programs which can be used either alone or part of a larger Basic program.

For example, a Basic loader program may require data written in hexadecimal notation to be converted to decimal before being POKEd into memory. In this case Listing 3.2 could be incorporated into the loader program.

Listing 3.1 **DECIMAL -> HEXADECIMAL**

```
100 REM CONVERTS DECIMAL NUMBER
110 REM TO HEXADECIMAL NUMBER
120 REM
130 INPUT"DECIMAL NUMBER";D
140 IF (D>65535)OR(D<-32768) THEN 130
150 H$=""
160 D=D+(D<0)*-65536
170 N=(D+(D>32767)*65536)AND(15)
180 H$=CHR$(48+N-7*(N>9))+H$
190 D=INT((D+(D<0)*-65536)/16)
200 IF D>0 THEN 170
210 PRINT H$
220 GOTO 130
```

Listing 3.2 **HEXADECIMAL -> DECIMAL**

```
100 REM CONVERTS UP TO 4 DIGIT
110 REM HEXADECIMAL NUMBER TO
120 REM DECIMAL EQUIVALENT
130 REM
140 INPUT"HEXADECIMAL NUMBER";H$
150 IF LEN(H$)>4 THEN 140
160 N=0
170 FOR I=1 TO LEN(H$)
180 V=ASC(MID$(H$,I,1))
190 IF (V<48)OR(V>70)OR((V>57)AND(V<65)) THEN 140
200 N=N*16+(V-55+(-7*(V<64)))
210 NEXT I
220 PRINT N,:IF N>32767 THEN PRINT N-65536 ELSE PRINT" "
230 GOTO 140
```

-ooOoo-

NOTES

CHAPTER 4

LOADING MACHINE CODE: PEEK, POKE & USR

Before we look at ways to mix M/C (machine code) and Basic programs, let's look at how we put the M/C bytes into RAM memory and then jump to that M/C program.

4.01 PEEK AND POKE

If we want to examine a particular memory location to see what value or byte is stored there, we can use the rather aptly named PEEK Basic command. (See page 103 VZ200 Basic Reference Manual, page 135 VZ300 Main Unit Manual).

```
PRINT PEEK(address)
```

will print to the screen the value of the byte held in the memory location specified by 'address'. Thus:-

```
PRINT PEEK(28672)
```

will return the value of the byte stored at location 28672 (7000H).

Note that 'address' must be the decimal equivalent of the address. Location 28672 decimal is the address of the top left hand position of the video screen. Try placing different characters at the top left hand position of the screen with the cursor and see what value is used to represent those characters in video RAM.

Notice that the values are within the range of 0 to 255. This is because this is the range of values that can be stored away in one 8-bit byte.

Of course, we can assign this PEEKed value to a variable, as in:-

```
A = PEEK(28672)
```

This command will work for both RAM and ROM (and indeed for all 65536 memory locations as it is a 'read' operation).

If we now want to store or alter a byte at a memory location, we can use another Basic command, POKE. (See page 103 VZ200 Basic Reference Manual, page 136 VZ300 Main Unit Manual).

```
POKE address,value
```

will place the number 'value' into memory location 'address'. Thus:-

```
POKE 28672,65
```

will put the value 65 into the top left hand corner of the screen which is the code for the letter 'A'.

Try POKEing values between 0 and 255 and note the different screen characters appearing in the top left hand corner. Compare these with the table of video screen Character Codes (See page 156 VZ200 Basic Reference Manual, page 204 VZ300 Main Unit Manual).

If you PEEK into the same location as you have just POKEd, you will find that the value you have POKEd is stored at that location. This is because the screen video memory is RAM (read and write).

If you attempt to store values at memory locations which are either empty or occupied with ROM you will find that this does not work.

In the case of empty memory space (areas without any memory chips) you will find a number between 0 and 255 (and it will most likely be different on consecutive PEEKs), whilst in the case of memory space occupied by ROM you will find that your value is not stored and each PEEK will return the same number (permanently stored in the ROM).

4.02 PLACES NOT TO POKE

Although it is possible to POKE into any address of the computer, it is very ill-advised to POKE into the Basic scratch pad between 7800H and 7AE9H, as this can easily cause a crash. Also, POKEing parts of the RAM which contain the Basic program lines or variables can result in the Interpreter losing track of the program, or the values of the variables being lost.

The Basic commands PEEK and POKE are extremely useful for entering and running M/C programs on the VZ. Without them, we would not be able to enter the bytes of our M/C programs into memory, and so we would not be able to run them.

Of course, this is overstating the case as we can use an Editor Assembler, but for small programs (and particularly for our initial experimentation) loading our M/C programs via Basic is simpler.

Now that we have a way of entering our M/C bytes from Basic we need a way of running them from Basic as well.

We need a Basic command which causes the CPU to leave the Interpreter, execute a machine code program that we have placed in some memory location, and then return to Basic.

There is a Basic command to do this. It is the USR function.

4.Ø3 USR FUNCTION

The Basic Interpreter, through the use of the USR function (page 141 VZ2ØØ, page 186 VZ3ØØ Basic Manuals), treats your M/C programs as machine code subroutines. The USR function call is a GOSUB call to a machine code program. The Interpreter does not actually execute a Basic GOSUB instruction, but a M/C equivalent.

Before executing a USR function call, you must POKE the starting address of your M/C program to a memory location in the Basic Interpreter's scratch pad memory space.

Because an address is a 16-bit number, there must be two 8-bit bytes to POKE. The USR function pointer location is 788EH and 788FH (3Ø862 and 3Ø863 decimal). It is a peculiarity of the Z8Ø that 16-bit numbers are stored in memory in reverse order.

Thus, if we wanted to store the address ØA ØØØH as our M/C program starting address, we would store Ø (ØØH) in location 788EH (3Ø862 decimal) and 16Ø (ØAØH) in location 788FH (3Ø863 decimal).

We will see later in Chapter 7 how we can pass values between the Basic Interpreter and M/C programs.

4.Ø4 USING PEEK TO EVALUATE SYSTEM POINTERS

The usefulness of the Basic PEEK command can be shown by using it to evaluate the System Pointers which reside in the communications region. (See Appendix 1 and 5).

The System Pointers have addresses stored in them in two-byte form, low order byte first, which specify various boundaries, such as the current top of memory, the end of the current Basic program text, start of variable storage, etc..

The first (low order) byte is the location within a page of memory, while the second (high order) byte contains the page number. To evaluate this as a memory location for PEEK the low order byte is added to 256 times the high order byte.

For example, to find the current top of memory (in 78B1/2H, see Appendix 1 and 5):-

First, evaluate 78B1H by looking up 78H and 0B1H in Appendix 7, which gives 78H = 120 decimal, 0B1H = 177 decimal.

Therefore 78B1H = 177 + 256*120 = 30897 decimal.

Now find the low order byte, and then the high order byte pointer value:-

```
LO = PEEK(30897)
HO = PEEK(30897+1)
```

Therefore the current top of memory = LO + 256*HO, or in one line:-

```
PRINT PEEK(30897) + 256*PEEK(30898)
```

Try finding the bottom of string space (78A0/1H = 30880/1 decimal) by the following:-

```
CLEAR 50
PRINT PEEK(30880) + 256*PEEK(30881)
```

Note that the bottom of string space should be 50 bytes below the current top of memory as set by the 'CLEAR 50' command line.

Now try:-

```
CLEAR 200
PRINT PEEK(30880) + 256*PEEK(30881)
```

Now the bottom of string space should be 200 bytes below the current top of memory.

Later chapters will show not only how to PEEK current pointer values, but how to POKE and update pointers as a means of reserving space for M/C programs. (See Chapter 7).

-ooOoo-

CHAPTER 5

THE Z-80 REGISTERS

5.01 THE Z80 MICROPROCESSOR UNIT (MPU)

The term microcomputer came into use to describe a computer system the next step down from the minicomputer, which in turn described smaller versions of the full-blown mainframe computer systems. A microcomputer system proper consists of a central processing unit (CPU), memory devices and various input/output (I/O) devices. The CPU itself consists of an arithmetic logic unit (ALU), some on-chip storage for the ALU to work in (called registers), and a control unit (CU) to keep all sections of the CPU working together.

A microprocessor chip like the Zilog Z80 combines all these elements of a MPU onto one chip. Although MPU is probably the more correct term to describe the Z80, it is more commonly called the CPU and that is the term used in this book.

5.02 INTERNAL REGISTERS

A register is simply an area of memory internal to the CPU which can be manipulated directly by the ALU in order to carry out arithmetic or logical operations at high speed.

These registers are either 8 or 16 bits wide, the width of the data and address busses respectively. Accordingly, the 8-bit registers are used mainly to store program data while the 16-bit registers are mainly used to store memory addresses.

There are ten 8-bit and four 16-bit registers in the Z80 CPU. Appendix 2 shows these registers and how they are organised.

Note how the 8-bit registers are grouped in 16-bit pairs. This is because these pairs of 8-bit registers are sometimes treated as 16-bit units as well as being accessible as individual 8-bit units.

5.03 THE ACCUMULATOR

The 'A' register or Accumulator is the primary register for arithmetic or logical operations. Most operations are best performed here, and sometimes it is the only place the CPU can carry out certain operations. This is where the result of an operation is stored or accumulated.

For example, if the number 7 is added to the contents of the accumulator (containing the number 2), the result (9) is written back into the accumulator (the original number 2 is lost). Additional information about the result of the last operation is contained in the 'F' or Flag register. This is why the 'F' register is shown paired to the A register in Appendix 2.

5.Ø4 THE FLAG REGISTER

This register is the only register in which the data cannot be interpreted as a number. Six out of the eight bits in the register are used to indicate, or 'flag', the result of an operation. Not all bits are affected by all operations, only those bits which are relevant to that operation.

The function of some of the bits is obscure, so we will initially confine ourselves to two bits: the ZERO and the CARRY flags.

The Z (Zero) and C (Carry) flags are bit 6 and bit Ø of the F register respectively.

The Z flag is used to indicate when an operation results in zero. This could be the result of subtracting two equal numbers, or the result of testing a single bit in the accumulator to see if it is a 1 or Ø. The important thing to remember is that the Z bit is set to 1 when there is a zero result and reset to a Ø when there is not a zero result.

That is:

Zero result (Z)	Z = 1
Non-Zero result (NZ)	Z = Ø

The C flag is used to indicate an overflow or underflow result. This can occur when the result of an addition is larger than eight bits (greater than 255). The C bit is set to 1 when there is a carry condition and set to Ø if there is not.

It is also possible to shift individual bits of a register or memory location through the carry bit which will be then be set (1) or reset (Ø) accordingly. Tests for the states of the Z and C flags allow different actions to be taken depending on the results of a previous operation.

5.Ø5 OTHER REGISTER PAIRS

The register pairs 'BC', 'DE', 'HL' are often called 'secondary accumulators' because many of the arithmetic and logical operations that can be performed on the A register can be performed on these registers as well.

These operations can be carried out the registers either as individual 8-bit or paired 16-bit units. They are sometimes called 'data counters' as they can be used to provide a pointer scanning through blocks of data in memory. The HL is the most useful of these as it can carry out many operations on memory locations that the other secondary accumulators cannot.

5.06 THE I AND R REGISTERS

These are two 8-bit special purpose registers which are rarely used by the software programmer. The 'I' (or Interrupt Vector) register contains half of a memory address used when the CPU responds to an interrupt input in Interrupt Mode 2 (IM2). The VZ is initialised to IM1 when switched on, so this register is normally not relevant.

The 'R' (or Refresh) register is more closely related to a hardware function of the Z80 than software. It keeps track of the address of dynamic memory currently being refreshed. What this means will be explained when we deal with memory devices in Chapter 6.

It is worth noting that the contents of the R register can be loaded in or out of the A register. You can use the constantly changing contents of the R register as a source of an integer random number between 0 and 255, or as the seed for a more complex random number generator program.

5.07 THE 16-BIT OR ADDRESS REGISTERS

The four 16-bit registers are single registers. They can only be manipulated as 16-bit units and are mainly used to store addresses and therefore are closely associated with memory access operations.

5.08 THE INDEX REGISTERS

The IX and IY registers are 'indexed addressing' registers, so called because they can be loaded with the base address (16-bits) of a block of memory. Instructions which then refer to these registers contain an offset (8-bit) from this base index address pointing to a particular byte within the memory block. The offset (or displacement) in the range -128 to +127 is added to the base address to reference the required byte.

5.09 THE STACK POINTER

The SP or Stack Pointer register keeps track of 16-bit data stored away temporarily in a specially reserved section of external memory.

The programmer must specify the starting position of this 'stack', as it is called, by loading a memory address into the SP register before the stack is used. The CPU can then store (by a 'PUSH' instruction) 16-bit numbers onto the stack which can be later retrieved by a 'POP' instruction.

The SP register keeps track of the next free memory space BELOW the last PUSHed data by being decremented after a PUSH operation and incremented after a POP operation.

The action of PUSH-ing and POP-ping is very similar to that of a letter spike used as a temporary file. The last letter spiked (PUSH) is the first available for access (POP). It is not possible to access a letter in the middle of the pile of spiked letters by a single operation. You need to alter the top of the pile by pulling off a bunch of letters in one go to uncover the buried letter.

Likewise, it is not possible to access a number stored in the middle of the stack by a single operation, but the top of the stack can be altered by loading a new address location into the SP register directly.

Believe me, this is not a prudent move, as the VZ video interrupt which occurs every 20mS uses the stack and can overwrite vital stack data. The interrupt can be disabled but if software routines in the Basic ROM are used the interrupt can be re-enabled causing all sorts of grief.

In general the SP register is best left alone once it has been initialised, except of course through the action of PUSH and POP.

Remember, the SP register is decremented after each PUSH operation, so, as 16-bit numbers are stored onto the stack it grows down in memory, i.e. upside down to the operation of our letter stack. The stack is also used when there is a call to a subroutine (like a GOSUB in Basic) because the CPU needs to store the return address so it can resume execution at that part of the program which called the subroutine.

5.10 THE PROGRAM COUNTER

The last 16-bit register is the PC or Program Counter. The CPU uses this register to keep track of where it is in the program that it is running. The program bytes are fetched from memory, one at a time, and the PC register is incremented to the address of the next byte which is required for program execution.

The only way the programmer can alter this register is indirectly, by causing program execution to continue at a different part of memory. This occurs when there is a CALL instruction (like the Basic GOSUB) or one of the several jump instructions (like the Basic GOTO).

5.11 THE ALTERNATE REGISTER SET

Just when you thought that the Z80 must be crammed with all these registers, now is the time to refer you back to Appendix 2. There you will note that the register pairs AF, BC, DE and HL are duplicated by another set of registers AF', BC', DE' and HL'.

These registers are called the ALTERNATE REGISTER SET. These registers are not accessible directly, however, the data contained in them can be accessed by exchanging the contents of the main registers with the contents of the alternate set.

The alternate registers seem like a good temporary store, but their use is not recommended unless you push them onto the stack before altering their contents. Although the Basic interpreter does not appear to use the alternate set, the Disc Operating System (DOS) does. All in all, I feel it best not to use them as temporary storage area.

-ooOoo-

NOTES

CHAPTER 6

HARDWARE - CPU, RAM, ROM & VIDEO

6.01 THE CPU BUSSES

There are three main groups of connections to the Z80 CPU: the 8-bit DATA BUS (data can flow in or out of the CPU via this bus), the 16-bit ADDRESS BUS (output only), and a mixture of input and output control signals which help the CPU synchronise with other devices. (Refer Appendix 3).

The data bus in the Z80 consists of eight 'two-way' digital lines which allows the CPU to read data in from, or, output data to, the rest of the microcomputer system. The two most common data transfers occur from the CPU to some external memory device (or vice-versa), and between the CPU and some I/O device.

The address bus conveys a 16-bit address output from the CPU. This address determines at which location within the microcomputer system data transfers will take place.

Note that the data bus is 8 bits 'wide', and the address bus is 16 bits 'wide'.

Looking at an 8-bit binary number, we should realise that the maximum value for that number occurs when all bits are 1. The maximum binary 8-bit number is then 11111111.

If you look carefully at a decimal or hexadecimal number you should notice that each digit has 'n' times the weight of the digit to its immediate right, where 'n' is the maximum count in each column. That is, 'n' = 10 for decimal and 'n' = 16 for hexadecimal.

As binary numbers have only two values (1 or 0) it follows that each digit in a binary number has twice the weight of the digit to its immediate right - starting from the right-most bit which has a weight of 1 - the decimal equivalent of 11111111 (the maximum value for an eight-bit binary number) is:

```
BINARY NUMBER      1     1    1    1   1   1   1   1
COLUMN WEIGHT  =  128 + 64 + 32 + 16 + 8 + 4 + 2 + 1
MAXIMUM VALUE  =  255
```

So the maximum range of numbers which can be represented by an 8-bit binary number is from 0 - 255.

Therefore, 0 - 255 is the maximum range for any single data byte in memory.

Instead of doing the addition of each digit weight as above, the maximum value could have been calculated by:

MAXIMUM VALUE = (2 ^ N)-1

where N = number of binary digits and '^' means 'raised to the power of'.

As N = 8, the maximum value = (2 ^ 8)-1 = 255, which gives 256 different values from 0 - 255.

We can use this equation to calculate the maximum number of different locations the 16-bit address bus can select by adding one to the maximum value.

MAXIMUM LOCATIONS = (2 ^ 16)

= 65536 (i.e. from 0 - 65535)

6.02 OTHER CPU INPUTS

The Z80 CPU is contained in a 40 pin chip. The data and address lines, plus power (+5v) and ground, account for 26 pins, leaving 14 pins for the third group of connections, the control signals. As mentioned previously, these signals allow the Z80 to communicate with outside devices in order to synchronise (or be synchronised with) external events. What this means exactly is best explained by examining some of these signals in more detail.

6.03 CLOCK INPUT

The most important control signal is the CLOCK input. The basic function of the clock is to provide a reference timing signal so that all the events which are taking place around the system can be synchronised. Also the rate at which events are completed is determined by the clock frequency.

In the VZ200 the frequency of the clock is 3,579,545 clock cycles (or periods) per second. This means that the smallest time interval used in determining the timing of events in the VZ200 is 1/(3,579,545) seconds (= 279.36511 nanoseconds). However, the smallest software timing interval is four clock periods or about 1.12 microseconds, as the shortest instruction takes four clock periods to execute.

In the case of the VZ300 the CPU clock frequency has been effectively shifted down from 3.5795 MHz to 3.5469 MHz.

Consequently program execution for the VZ300 is 0.9% slower, and the frequencies generated by the SOUND command are 0.9% flat when compared to the VZ200. This timing difference should be accounted for when using critical software timing loops in your M/C programming.

6.04 RESET INPUT

This input to the Z80 CPU is activated every time the power is switched on. Its purpose is to reset the Z80 to certain initial states so that execution can begin from a known starting point.

The input is what is called an 'active low' input. The term 'active low' refers to the input only having any effect when it is taken to a logic low (0) level.

There is a external delay circuit connected to this input which holds the level to a logic low long enough for the power supplies to the rest of the circuit to have stabilised after switch on. After this delay the reset input is taken to logic high (1) and execution begins from location 0000H.

6.05 INT INPUT

This input allows an external hardware device to 'interrupt' the normal program flow and force the CPU to execute a different piece of machine code. When finished, the CPU resumes executing where it left off from the normal program flow.

The other eleven control signals are not directly applicable to the software programmer as they relate to purely hardware considerations.

All Z80 pin connections are available at the 'memory expansion' interface at the back of the case (except BUSRQ and BUSAK). These are not buffered so care must be exercised when any connections are made to them to avoid permanent damage to the internal chips.

6.06 MEMORY DEVICES

As mentioned previously, the address bus inside the VZ is 16-bits wide which gives the Z80 access to 65536 separate memory locations in which to store data. In case you're wondering why you have heard Z80 based computers being able to address 64K bytes of memory, not 65.536K bytes, the answer is that '1K' bytes of memory is 1024 bytes and not 1000 as you might expect (1024 = 2^10). Divide 65536 by 1024 and you will come up with 64K.

There are two types of memory devices in the VZ; these are called Random Access Memory (RAM) and Read Only Memory (ROM). The terms RAM and ROM have these days become less than clearly descriptive, RAM originally used to distinguish between core memory (where any byte could be accessed in any random order) and hardcopy memory devices such as paper and magnetic tape (where bytes of information had to be accessed sequentially).

6.07 RAM MEMORY

The term RAM has now been pressed into service to refer to memory from which we can not only read, but also write new data. It therefore could be more accurately termed Read/Write Memory. This is the memory in which data that must be able to be changed, are stored.

Examples are:- variables that are changed while the program is running, program listings while writing Basic programs, or, the memory which contains the everchanging video display information.

While a very useful component, RAM has one very inconvenient characteristic. When the power is removed completely, the memory loses all the data it held. This means we need to store our programs on tape or disc to keep permanent copies of them. This is why the Basic Interpreter is stored in permanent memory (ROM), so that it is available as soon as the VZ is switched on.

Some computers have small batteries connected across their static RAM chips which continue to supply power to the chips even when the main power supply has been switched off. The VZ is not one of them.

6.08 STATIC AND DYNAMIC RAM

MOS (Metal Oxide Semiconductor) RAM memories are made in two basic types, STATIC and DYNAMIC. In MOS static RAM the bits are stored in MOSFET Flip-Flops called latches. It is referred to as static because the state of the latch (binary data) is retained as long as power is supplied. The static RAM chips used in the VZ are type 6116 which contain 2K (2048) byte locations each with 8-bits.

The standard VZ200 has eight of these static RAM chips, giving 8K bytes of static memory for the unexpanded VZ200. Of this 8K bytes 2K is used for the video RAM, plus 745 bytes for the Basic Scratch Pad and 50 bytes for the initial string space. This leaves about 5349 bytes available for actual program RAM space.

The VZ300 has just one 6116 static RAM chip as standard which is used for the video RAM, the rest of the 18K bytes total memory being provided by in-built dynamic RAM.

MOS dynamic RAM stores data bits in small capacitors instead of flip-flops as in static RAM. This obviously is a very simple cell arrangement and allows very large memory arrays to be constructed on a chip.

The relative physical size of typical static and dynamic RAM chips illustrates this point vividly. The 4116 dynamic RAM chip is about one-third the size the size of the 6116 static RAM chip. Both hold 16K bits of data.

Unfortunately, the capacitor used in the dynamic memory chip is so small that it quickly loses it charge and the data will 'fade' unless it is re-written, or refreshed, at least every 2mS.

With most types of CPU this requires additional circuitry and complicates the operation of the dynamic memory. However, the Z80 has additional circuitry built in to take care of this. The R register is part of this inbuilt circuitry. The refresh operation is done transparently to the operation of the Z80 during the normal time for fetching and execution of instructions. Very neat !

6.09 ROM MEMORY

The term ROM, or Read Only Memory, is more descriptive as you can only read from these memory locations during normal operation. This memory is used to store data of a fixed nature because the data is retained when the power is switched off and cannot be altered by the programmer, even using machine code.

For example, in the VZ, the machine code which implements the Basic Interpreter (memory locations 0000H to 3FFFH = 16K bytes) is held in ROM - as is the machine code which runs the Disc System (the ROM is in the Disc Controller Interface and occupies memory locations 4000H to 5FFFH = 8K bytes).

Every computer needs a minimum amount of program code (called a Bootstrap Program) when it is switched on because the CPU starts executing almost immediately, and from then on needs code to execute continuously. It is convenient to store this bootstrap code in ROM.

The bootstrap code in the VZ is actually the initialisation routines of the Basic Interpreter ROM.

6.10 THE 6847 VIDEO DISPLAY GENERATOR

The Motorola MC6847 Video Display Generator (VDG) reads data from the video RAM and produces the video signal which allows the generation of alphanumeric and graphic displays. The chip is capable of four different alphanumeric display modes, two semigraphic (low-resolution) modes, and eight graphic (high-resolution) display modes.

Before you become too excited about this, the four alphanumeric modes are normal,inverse,green or orange background; the semigraphic mode is locked into what is called the SEMIGRAPHICS FOUR mode (four blocks per character space, eight colours); the graphics mode is locked into COLOUR GRAPHICS TWO mode (graphics only, with 128 blocks or 'pixels' across and 64 pixels down, four colours).

The display modes that are available from Basic are fixed by hardware connections and are not alterable by software.

The VDG provides all the signals required to produce a colour display including vertical and horizontal sync pulses, chrominance (colour) signals and luminance (brightness level) signals. These signals are combined externally and are accessible as composite video signals (for direct connection to a monitor), or as a modulated VHF signal at the rear of the VZ suitable for connection to the antenna terminals of a domestic TV set.

The VDG also provides a hardware INT (interrupt) signal to the CPU to tell it when the vertical retrace period has started. This allows the CPU to access the video memory at times when the VDG is not writing video data to the screen, avoiding undesirable flicker.

The VDG scans through the video RAM memory and reads out the character codes that have been stored there. It then switches the video level at the correct times as the screen is scanned by the electron beam to produce the outline of the character on the screen. In fact the process is in principle the same as the way a dot-matrix printer produces a character on a page by firing needle hammers into the printer ribbon at the correct times. The shape of the outline of the dots closely spaced gives the impression of a continuous character outline.

When the VDG is in the graphics mode (MODE (1)) the VDG scans the full 2048 bytes of the screen RAM memory. In the text mode (MODE (0)) the VDG only scans the first 512 bytes of the video memory.

-ooOoo-

CHAPTER 7

BASIC & MACHINE CODE - PEACEFUL CO-EXISTENCE

7.Ø1 COMBINING MACHINE CODE WITH BASIC

For some reason or other, when the move is made into machine code, programmers seem to abandon Basic completely. In general programs are machine code only. I am not convinced that this is the best approach when using M/C in a computer which already has a Basic Interpreter resident in ROM.

Certainly there is a certain sense of satisfaction in being able to show you can code your program entirely in machine code, but surely the aim should be to write your programs efficiently, quickly and (as is sometimes forgotten) in a form which can be debugged easily. I consider this to be a professional approach to programming.

As well, it is important to develop your M/C programs in such a way as to allow them to be easily modified for use in other programs. This results in substantial savings in time and effort in your programming activities.

The general approach I would recommend is to use Basic programming as much as possible (i.e. only use M/C where speed is of paramount importance, or it simply not possible to carry out an operation using Basic) and then use the Basic USER function call to run M/C routines for those areas where M/C is required.

For example, if you were writing a program to read in programs off tape without executing them, (to find out information about the program contained in the header), you could use Basic for prompts, instructions or menu displays, only using M/C routines (called from Basic) to read the data from the tapes.

The strength of this approach is the ability to save the M/C routines by just saving the Basic program, being able to edit the M/C program using the standard Basic screen editor, and saving heaps of time trying to write machine code routines for operations for which Basic would be just as suitable, i.e. where speed is not important.

There are disadvantages of course, the main one being that some instructions in your M/C program may refer to absolute memory locations and so the program cannot be readily moved around and made to run at a different location in memory.

Of course, this occurs even when you are using M/C code only, but then you are usually working through a utility program called an Editor Assembler which makes all the necessary adjustments to the program code automatically.

When you are using Basic to load in the M/C and then execute it, and you want the machine code to execute at a different location, all absolute jumps must be recalculated and updated by you. The solution to this is to make all your machine code relocateable using RELATIVE reference instructions.

All this may not be clear to you at this stage, as we have not looked at the different types of instructions yet, but will become clear later.

7.Ø2 SAFE PLACES FOR MACHINE CODE

If we are going to make our M/C programs as routines callable and co-resident with Basic programs we need to find a safe place to store them. Obviously this has to be in RAM memory, because, as you will remember, it is not possible to alter ROM memory space. To ensure this, we obviously need to know where RAM and ROM memory resides.

Some areas of RAM are not suitable because the Basic Interpreter uses them to store information it needs to keep on executing correctly. As the Basic program runs it may use extra variables which require extra memory space to store them. The Interpreter also uses a section of memory for its own 'stack', a place it uses as a temporary store. Indeed, it uses the stack to store the return address at which the CPU needs to continue after being directed to execute our M/C routines via the Basic USER function.

If we are not careful where we place our M/C routines, there is a good chance they will be overwritten by the normal run of events during execution of the Basic Interpreter.

At best, this will result in wrong results from our program and at worst, cause a 'crash'. This is where the CPU gets out of step with the intended program flow and blindly tries to execute what it 'thinks' is the correct sequence of instructions. This can often only be stopped by resetting the computer to regain control. In order that we do not put our M/C programs in these expanding areas, we need to know which areas in RAM the Basic Interpreter uses.

A memory map is used to illustrate the organisation of memory space within the computer, but before we look at this, we will have a closer look at the Basic Interpreter itself and briefly how it operates.

7.Ø3 SO YOU THINK YOU'RE A PROGRAMMER ?

As we write our Basic programs, we slip into the false notion that we are writing Basic program 'code'. In fact, the only code used while a program runs is the original Basic Interpreter code which came with the machine (resident in the Basic ROM) when we bought it.

The program we write in Basic is, in reality, a linked data list which the Basic Interpreter machine code uses to carry out the operations as specified in that program. As the Basic program runs, the Interpreter reads the data from the Basic program data list which tells it which sections of machine code to execute in the Basic ROM to carry out the actions of the Basic program.

When we write our machine code routines and call them from Basic we are, in effect, extending the Interpreter, and should aim to integrate those routines with the operation of the Interpreter as much as possible.

7.Ø4 VZ MEMORY MAP

Appendix 1 shows a representation or 'map' of how the unexpanded VZ200/300 is organised. Recall that the address of a byte in memory is the location of that byte. Each memory location can store an 8-bit byte which in turn represents a number between Ø and 255. The addresses in Appendix 1 are given in hexadecimal and decimal notation. The map shows how the memory is used for Basic programming.

7.Ø5 BASIC ROM

The first 16K (16384) bytes of memory is the Basic ROM, occupying memory address ØØØØH to 3FFFH. This area is obviously unusable for the storage of M/C programs.

The VZ 16K Basic is essentially TRS-8Ø Level II Microsoft Basic (12K) plus 4K bytes of enhancement (sound, inverse text handling, high-res graphics and different cassette routines). That is not to say that the Basic is more comprehensive than Level II, but the commands implemented are essentially the same as Level II.

Sure, the VZ does not compare with the more sophisticated microcomputers (neither does it have a sophisticated price tag), but viewed from the angle of what it was obviously intended (a way of getting your feet wet without emptying your pockets), it is close to ideal.

Because the VZ Basic is largely identical with that used in the TRS-80 and System 80, anyone who has owned one of these machines has a head start (especially on me, as I have never even used one).

Much of the TRS-80/System 80 software (both Basic and M/C) can be adapted. A M/C utility program has been written by the author which re-enables all the Level II Basic commands hidden away in the VZ Basic which allows many TRS-80/System 80 Basic software listings to be adapted (except graphics of course).

7.06 ROM CARTRIDGES

The area of memory from 4000H to 67FFH has been reserved by the designers for the addition of extra software in ROM cartridges, occupying the area between the Basic ROM and the memory-mapped I/O (input/output) area from 6800H to 6FFFH. This does not mean that you cannot use this area for other purposes (extra RAM, more memory-mapped I/O etc.), but the Basic Interpreter checks for the presence of ROM cartridges starting at 4000H, 6000H and 8000H (during initialisation upon switch-on) and if it finds the correct byte sequence at one of these locations, execution jumps to that ROM cartridge. This is how control is passed to ROM cartridges such as the Serial Interface and the Disc Drive Interface.

7.07 MEMORY-MAPPED I/O

The memory mapped I/O from 6800H to 6FFFH is the interface for the keyboard, cassette, video display controller and sound circuits. The I/O hardware interface circuits are called memory-mapped because I/O operations are normally carried out by special I/O machine code instructions 'IN' and 'OUT' through 'I/O ports'.

Memory-mapped I/O allows access to hardware circuits through memory operations which can allow simplification of both software and hardware.

An example of this is the keyboard scanning interface circuit, which we will deal with in more detail in Appendix 4.

7.08 VIDEO SCREEN RAM

Memory space from 7000H to 77FFH is occupied by 2k bytes (2048 bytes) of video screen RAM (See Appendix 1). In MODE(0) only the first 512 bytes of this memory space is used, organised as 32 characters across by 16 lines down, each character requiring one byte (32 * 16 = 512 bytes).

During MODE(1), all 2048 bytes are used to give a high-res screen of 128 pixels (points) across by 64 pixels down. There are 4 horizontal pixels/byte, therefore there are 32 bytes across by 64 down (32 * 64 = 2048 bytes).

7.09 BASIC SCRATCH PAD RAM

The Basic Interpreter needs reserved space to use as temporary storage of values and system pointers when it is executing. This area can be loosely defined as occupying addresses 7800H to 7AE9H. I say loosely because not all addresses are used in this area.

Address 7AE9H is the start of Basic program area. For more details on System pointers (start of Basic, end of Basic, etc.) see Appendix 5.

The Basic Scratch Pad area should not be disturbed as the results can be disastrous. Although there are a few areas which appear not to be used by the Interpreter, I prefer to recommend that you use other areas for your M/C programs.

From the above we can see, for the VZ, the area that we can put our M/C programs is from 7AE9H to the top of memory (which varies according to how much RAM is present).

I have, on occasions, placed M/C programs in the unused top of video RAM (7200H to 77FFH when in MODE(0)), but this was only when I had no other room to place them. If you do use this portion of screen RAM then you must be sure that you do not enter MODE(1) otherwise you will lose your code.

In any case because the CPU is accessing the video memory chip during execution of a program in this area, a flicker will be produced on the screen.

The area 7AE9H to top of memory is also occupied by Basic programs, and so we need a method of making Basic and M/C programs mix together without causing conflicts.

7.10 HOMES FOR M/C PROGRAMS

The safest place for M/C programs is in memory which has been stolen away from that which is available to Basic programs. If we can fool the Basic Interpreter into taking the top of available memory to be lower than what is actually available, we can use this memory space above the new top of memory to store our M/C programs.

The Basic Interpreter stores the pointer to top of memory (TOM) in locations 78B1H and 78B2H (30897 and 30898 decimal).

We can alter this from Basic (carefully!) by the program in Listing 7.1.

The value TM contains the new TOM and so we can put our M/C program in memory starting from the next byte up from this new TOM (and all the way up to the old TOM) without interfering with the operation of Basic.

Note that this method is independent of memory size.

Listing 7.1

```
100 RB=100: REM NUMBER BYTES TO RESERVE
110 TM=(PEEK(30897)+PEEK(30898)*256)-RB: REM NEW TOM
120 MS=INT(TM/256):LS=TM-MS*256: REM FIND MSB & LSB
130 POKE 30897,LS:POKE 30898,MS: REM ENTER NEW TOM
140 CLEAR 50: REM RESET ALL POINTERS
150 TM=PEEK(30897)+PEEK(30898)*256: REM FIND NEW TOM
160 PRINT TM
```

If you add the following line:

```
170 GOTO 100
```

you will see successive TOM values printed out on the screen until the amount of available space for Basic is reduced to a level so small that there is not sufficient space for the program itself to operate.

This results in an **'OUT OF MEMORY ERROR'** in line 150.

7.11 EMBEDDING MACHINE CODE IN BASIC

Now that we can reserve memory for our machine code programs, POKE bytes into that memory and then jump to the start of our machine code, we need a method of actually storing our machine bytes in Basic so that they can be easily loaded.

The first method of embedding machine code is to include the code in DATA statements and then move this to a fixed location in memory. Let's see how this works on a simple program.

Listing 7.2 contains a short program which scrolls the screen **DOWN** by one line.

For the moment don't worry about how the actual M/C code program works. Our aim is to learn how to load M/C programs from Basic. However, you might like to refer to Chapter 11 for clues.

Listing 7.2

LINE	LABEL	MNEMONIC		HEX CODE	DECIMAL
001	SREG	PUSH	AF	F5	245
002		PUSH	BC	C5	197
003		PUSH	DE	D5	213
004		PUSH	HL	E5	229
005	SCRL	LD	HL,71DFH	21 DF 71	33,223,113
006		LD	DE,71FFH	11 FF 71	17,255,113
007		LD	BC,1E0H	01 E0 01	1,224,1
008		LDDR		ED B8	237,184
009		LD	HL,7000H	21 00 70	33,0,112
010		LD	A,60H	3E 60	62,96
011		LD	B,20H	06 20	6,32
012	LOOP	LD	(HL),A	77	119
013		INC	HL	23	35
014		DJNZ	LOOP	10 FC	16,252
015	RREG	POP	HL	E1	225
016		POP	DE	D1	209
017		POP	BC	C1	193
018		POP	AF	F1	241
019		RET		C9	201

If you look at the previous listing, you will see that the decimal equivalent of each hexadecimal byte has been worked out by the following: e.g. for hexadecimal '0F5H'

```
DECIMAL = (F * 16) + (5 * 1)
        = (15 * 16) + (5 * 1)
        = 240 + 5
        = 245
```

Also note how the order of the two 8-bit bytes which make up the 16-bit values to be loaded into the HL, DE and BC registers are reversed in order in memory (see lines 005,006,007 and 009).

This is the case for all 16-bit values in memory when they are to be used by the Z80.

So now it is simply a matter of entering the decimal equivalents into DATA statements in order that they can be loaded into memory from Basic via the use of POKE.

Firstly, we must allocate the memory space required (31 bytes), but say we allocate 40 bytes of memory. We use the TOM resetting program as before, i.e.:- (see over page).

Listing 7.3

```
100 RB=40                                          :REM RESERVED BYTES
110 TM=(PEEK(30897)+PEEK(30898)*256)-RB :REM NEW TOM
120 MS=INT(TM/256):LS=TM-MS*256                    :REM FIND MSB & LSB
130 POKE 30897,LS:POKE 30898,MS                    :REM ENTER NEW TOM
140 CLEAR 50                                       :REM RESET POINTERS
150 TM=PEEK(30897)+PEEK(30898)*256                 :REM FIND NEW TOM
160 ST=TM+1                                        :REM RESERVED AREA
```

Next, we load the decimal machine code bytes from DATA statements into the reserved area.

```
170 FOR J = 0 TO 30                                :REM 31 BYTES
180   READ D                                       :REM READ BYTE
190   L=ST+J                                       :REM NEXT LOCATION
200   IF L>32767 THEN L=L-65536                    :REM RANGE OF POKE
210   POKE L,D                                     :REM BYTE INTO MEMORY
220 NEXT J
```

The actual DATA statements are put at the end of the program (my preference).

Note how the location to which the byte is to be loaded is checked in line 200. This is because the two Basic memory reference functions PEEK and POKE (also the FOR...TO...NEXT loop index) won't accept 16-bit integer values greater than 32767. If you reference memory locations in the upper 32K of memory (above 32767) then the location must be entered as:

```
location = location - 65536
```

If you don't then an 'overflow' error results.

Next, the USR function needs to know where the start of the M/C program is. The variable 'ST' contains the decimal equivalent of the start of the M/C program. It is a 16-bit address (in the range 0-65535) and the MSB (most significant byte) is evaluated by dividing by 256. The LSB is found by multiplying the integer value of that division by 256 and subtracting from the original value, i.e.:-

```
230 MS = INT(ST/256)
240 LS = ST-(MS*256)
```

For example, let's say 'ST=50000'

```
MS = INT(50000/256) = 195
LS = 50000 - (195*256) = 50000 - 49920 = 80
```

Working back the other way to check.

ST = (MS*256) + LS = (195*256) + 80 = 50000

We put these two bytes into the special USR function pointer (remembering to put them in reverse order in memory) located at 788EH and 788FH (30862 and 30863 decimal).

```
250 POKE 30862,LS: POKE 30863,MS
```

Provide a delay to slow the downward scroll action (because the M/C action is too fast).

```
260 CLS
270 PRINT@0,RND(0)*10000000
280 FOR D=1 TO 100:NEXT D
```

Now the USR function call can be used to jump to the M/C routine. We will call it straight away in our demonstration program here, but in an actual program it can be called any time after it has been loaded into memory, and the USR function pointers at 788EH and 788FH have been loaded correctly.

```
290 X=USR(0)
```

Go back and print a new line and scroll down.

```
300 GOTO 270
```

Now the DATA statements.

```
310 DATA 245,197,213,229,33,223,113,17,255,113,1,224
320 DATA 1,237,184,33,0,112,62,96,6,32,119,35,16,252
330 DATA 225,209,193,241,201
```

Type in the above Basic program, CSAVE it (or SAVE it if you have a disc system) and then RUN it.

You should see a random six digit number being printed at the top of the screen and then the whole screen moved one line down.

So to embed M/C in a Basic program by this method do the following:

1. Write the program in mnemonics and trap check it.
2. Write down the hexadecimal code for the mnemonics.
3. Convert the hexadecimal values for the machine code into their decimal equivalents.
4. Put these decimal equivalents into DATA statements in the Basic program.
5. Use the Basic program to allocate memory space for the M/C by lowering the TOM pointers for Basic, leaving space above the new TOM. Reset Basic pointers.
6. Move the DATA values to this reserved space above the new TOM.
7. Set up the USR address (30862 and 30863) and call the machine code as required.

7.12 DISADVANTAGES OF 'DATA STATEMENT' METHOD

Because the actual position of the code is determined by the amount of memory resident in the computer in which the program is to be loaded, the M/C program must not contain absolute memory address references to within the program itself.

For example, if you wanted to load the HL register with the beginning of a text message block within your program, you are in trouble. This is because this absolute memory address reference depends on where the program is loaded.

Of course, if the absolute address is outside the program (e.g. CALL 3450H) there is no problem because this is in ROM and so does not change position when your M/C program is loaded into different places in memory.

The actual amount of memory varies between VZ200 and VZ300, or when a 16K memory expansion is fitted, or indeed if there is the Disc Drive Controller fitted (The Disc Operating System grabs 300 odd bytes from the top of memory and lowers the 'Top of Memory' pointer when the VZ is switched on).

If you want to write a M/C program loaded from DATA statements that will work on any VZ you can do one of three things:-

1. Write your code without the use of absolute memory address reference instructions, not a simple task !

2. Include in your Basic loader an 'absolute address adjusting' section which corrects all the absolute memory references according to the the actual loading address of the M/C program due to different memory sizes. This is the method I used for my Extended Basic utility, although the 'address adjusting' routine was part of the M/C program itself. Certainly not an easy task !

3. Decide on a fixed top of memory value which will work for all variations of memory size. For example, say you want your program to work for an unexpanded VZ300 or VZ200 with 16K memory expansion. Referring to Appendix 1, choosing a fixed top of memory value of 0AFFFH (45055 decimal) would allow your M/C code program to lie between 0B000H (45056 decimal) and 0B7FFH (47103 decimal). This allows just over 2000 bytes for the M/C program.

Notice that this wastes space in the case of an expanded VZ200 (and indeed an expanded VZ300), but this is the penalty for the simplicity of having a fixed location to load your M/C program, that is, it can contain absolute memory address references to within itself. This simplifies the programming process considerably.

Incidently, a number of M/C programs originally written for the 16k VZ200 use this fixed loading address technique (although not loaded from a Basic loader, but a M/C tape directly). This is why they will not work with an unexpanded VZ300, because the top of memory for the unexpanded VZ300 is lower than an expanded VZ200 (Refer Appendix 1).

As an extra trap, remember that in the case of the unexpanded VZ300, almost 400 bytes has been grabbed from 0B7FFH down by the Disc Operating System if it is fitted. This means that if the program is to work on an unexpanded VZ300 then only about 1500 bytes should be used above the new top of memory.

Given the above limitations, this method still has the strong point of simplicity. To reserve 2047 bytes for a program to work on an unexpanded VZ300 or and expanded VZ200 the following simple process is used:-

```
1 POKE 30897,255:POKE 30898,175:CLEAR 50
```

This will set the 'Top of Memory' pointer to 0AFFFH (45055 decimal), allowing about 1500 bytes up from 0B000H (45056 decimal) to be used to load the M/C program.

The **'CLEAR 50'** command is needed to reset all the other Basic pointers which are set relative to top of memory.

If you require more memory than this simply POKE location 30898 with a lower value than 175. Lowering this value by 1 lowers the top of memory by 256 bytes.

For example, to set the top of memory to 8FFFH (36863 decimal), POKE location 30898 with 143 instead of 175. Now memory from 9000H to 0B7FFH (36864 to 47103, reserving just over 10000 bytes), is available for your M/C program, less about 400 bytes down from 0B7FFH if a Disc Controller is fitted.

Of course, now you have only memory from 7AE9H (31465 decimal) to 8FFFH (36863 decimal) available for the Basic program itself.

7.13 MACHINE CODE EMBEDDED IN 'REM' STATEMENTS

For small programs (maximum of about 60 bytes) we can put our M/C program in a fixed area which does not change with memory size. This allows the use of absolute memory references because the location of the M/C program is known. We do this by placing our machine code in the location in memory occupied by a REM statement.

By making the REM statement the very first line (say line number 0), and because the start of Basic programs does not change with different memory sizes, we have a fixed address to load the code.

The major restrictions are size (less 60 bytes) and the fact that you can't have a zero byte in the code (because Basic treats this as an end of line marker and will get terribly confused if it finds one before the real end of line when you try to edit the Basic program). There are some sneaky ways around this as we can see in the modified scroll routine in Listing 7.4.

Note that there was a zero byte in the previous M/C program (line 009 in the machine code listing). To avoid this zero byte in the modified program above, we load HL with 6FFFH and then INCrement it by one to 7000H in the next line (see lines 009 and 010). We now can load this code into a REM statement.

First we create the REM statement (do NOT use the REM short form apostrophe as it takes two bytes to store it in memory as against one for the standard REM).

```
Ø REM*****THIS TEXT MAKES SURE THAT SUFFICIENT SPACE IS MADE*****
```

The start of where Basic programs are stored in memory is 7AE9H (31465 decimal). Basic uses two bytes (31465-31466) to hold the address of the next Basic line, another two bytes (31467-31468) to hold the line number of this line, and one byte (31469) to hold the token for REM.

Therefore we can begin loading our machine code at location 31470 (7AEEH).

Listing 7.4

LINE	LABEL	MNEMONIC		HEX CODE	DECIMAL
ØØ1	SREG	PUSH	AF	F5	245
ØØ2		PUSH	BC	C5	197
ØØ3		PUSH	DE	D5	213
ØØ4		PUSH	HL	E5	229
ØØ5	SCRL	LD	HL,71DFH	21 DF 71	33,223,113
ØØ6		LD	DE,71FFH	11 FF 71	17,255,113
ØØ7		LD	BC,1EØH	Ø1 EØ Ø1	1,224,1
ØØ8		LDDR		ED B8	237,184
ØØ9		LD	HL,6FFFH	21 FF 6F	33,255,111
Ø1Ø		INC	HL	23	35
Ø11		LD	A,6ØH	3E 6Ø	62,96
Ø12		LD	B,2ØH	Ø6 2Ø	6,32
Ø13	LOOP	LD	(HL),A	77	119
Ø14		INC	HL	23	35
Ø15		DJNZ	LOOP	1Ø FC	16,252
Ø16	RREG	POP	HL	E1	225
Ø17		POP	DE	D1	2Ø9
Ø18		POP	BC	C1	193
Ø19		POP	AF	F1	241
Ø2Ø		RET		C9	2Ø1

Remember we have an extra byte to load (32 bytes now) because of the trick to eliminate the zero byte out of the original program.

Listing 7.5

```
1ØØ FOR J = Ø TO 31                    :REM 32 BYTES TO LOAD
11Ø   READ D                           :REM READ BYTE
12Ø   POKE 31470+J,D                   :REM PUT BYTE INTO MEM
13Ø NEXT J
```

Now the DATA statements.

```
140 DATA 245,197,213,229,33,223,113,17,255,113,1,224
150 DATA 1,237,184,33,255,111,35,62,96,6,37,119,35,16,252
160 DATA 225,209,193,241,201
```

We can RUN this program as it stands now, so CSAVE it first and then RUN. Now LIST the program again. You will notice something strange has happened to the REM statement in line number 0. The text has been replaced by the single Basic command 'VAL'. This is because decimal 245 (the first value in the DATA list) is the Basic token for the Basic command 'VAL'.

Nothing else appears on the line because the next byte in the line has a value (197) which the Interpreter cannot recognise as a valid text character (excluding graphic characters) or a valid command token, so it just skips to the next line and continues listing.

Nevertheless, our M/C bytes are still stored away in that line in memory (even though the Basic LIST function doesn't want to admit to the fact!), and can be saved just as if it was ordinary text within a REM statement.

In fact, we no longer need lines 100 - 160 and they can be deleted (assuming of course that we don't want to alter the M/C program at some other time). So we are just left with line 0 which has our small M/C program hidden away.

We need only now set up the USR function by POKEing the start address of the M/C program (31470) and then we can access the code at any time by using the USR function call.

Because the location of the start address (31470) is fixed it might be worthwhile to work out the two bytes that have to be POKEd into the USR function pointers and note them down.

```
MSB = INT(31470/256) = 122
LSB= 31470 - (MS*256) = 31470 - (122*256) = 238
```

If you possess a calculator that does decimal/hexadecimal conversions then first convert 31470 decimal to hex:-

```
31470 = 7AEEH
```

Split this 16-bit hex number into two 8-bit bytes:

```
LSB = EEH, MSB = 7AH
```

Then convert each byte back to decimal:-

LSB = 238, MSB = 122

Therefore, before you use the USR function call, set up the pointers by:-

```
200 POKE 30862,238:POKE 30863,122
```

Then access the M/C at any time by:-

X=USR(0)

It is always wise to set up the USR function pointers every time we want to call our machine code, just to ensure that they are correct. If the pointers have been altered inadvertently, the program execution may continue in some unexpected (and inconvenient) direction.

7.14 M/C BETWEEN THE END OF BASIC PROGRAM AND ITS VARIABLE STORAGE AREA

A third method of reserving space for a M/C program is to artificially raise the start of the variable storage area which normally rides directly on the top of the end of the Basic program text.

This method requires careful thought to prevent either the variable storage area, the Basic program text itself or the M/C from interfering with each other. The idea is to use the space between the end of Basic program text and the upward moved variable storage area to locate our M/C program.

To move the variable storage upward we tell the interpreter that the Basic program text is longer than it actually is by altering the 'End of Basic' pointer located at 30969 and 30970 decimal (78F9H and 78FAH).

Firstly, find out where your Basic program text actually does end by typing in:-

PRINT PEEK(30970)*256 + PEEK(30969)

Write this value down. Next determine how long your M/C program is and round off to the nearest next multiple of 256. That is, if the length of your M/C program is 680 bytes, the next nearest multiple of 256 is 768 (3*256). Then adjust the MSB of the 'End of Basic' pointer by this amount. This must be done in the very first line of the Basic program:-

```
1 POKE 30970,PEEK(30970+3)
```

This will raise the apparent end of Basic by 3*256 = 768 bytes. We can now slot a M/C program between the real end of Basic (the value written down above) and our artificial end of Basic which is 768 bytes further up in memory.

Actually, you should go for gross overkill on allocating space because it is only after you have typed in all the Basic program can you be sure of the length of Basic text. However, if you are loading the M/C program from DATA statements, you cannot finalise the code for the M/C program which refers to absolute memory addresses before you know where the Basic text ends - Catch 22 !

This is why I prefer lowering the 'Top of Memory' pointer to reserve space as the actual location of the machine code does not need to change if the Basic program grows by a significant amount.

There are other traps in the end of Basic approach, and they are:-

1. Moving the end of Basic too high up in memory can leave too little space for the variable storage area which grows up from this point and the Basic stack area which grows down from the top of memory.

2. Be careful that any editing of the Basic program does not cause the real end of Basic text to increase and encroach on the M/C program loading area because if you are loading the M/C from DATA statements (or any method which loads the code after the Basic program is loaded) it is possible to inadvertently overwrite the Basic text.

We now have the capability of calling machine code programs while in the middle of Basic program execution. This means that time-critical parts of the Basic program can be coded in machine code to speed up program execution, while Basic can be used for operations which are harder to code in assembly language, such as string input and handling, printing on the screen, and number operations (number-crunching).

In summary, the procedure for calling M/C programs from Basic via the USR call is as follows:

1. POKE the address of the start of the machine code routine to be called into the USR pointers at 30862 and 30863. The two bytes represent the least significant and most significant bytes of the address in standard Z80 16-bit format (least significant first then most significant).

2. Any time a call to the machine code routine is required, call the routine by a Basic USR function call such as X=USR(0).

You can make calls to more than one M/C routine by setting up the USR pointers 30862 and 30863 to the proper address just before the USR call.

7.15 PASSING VALUES BETWEEN BASIC AND MACHINE CODE ROUTINES

The USR call function has two arguments:

```
X=USR(0)
```

X and 0 are two dummy arguments that can be used to communicate 16-bit integer values between Basic and your machine code routines. The argument inside the brackets can be sent to the machine code routine and a value can be returned from the machine code in the variable X.

In fact, I sometimes prefer to write the USR function with the following dummy parameters:

```
R=USR(S)
```

where 'S' signifies data Sent to the machine code routine and 'R' signifies data Received. When the above USR call is made, the 16-bit value of 'S' is loaded into 31009 and 31010 (7921H and 7922H) where it can be accessed by your machine code routine.

On return from the machine code program (caused by a RET instruction at the end of your machine code routine), the contents of 31009 and 31010 will be assembled into a 16-bit integer value and loaded into 'R', where it can be used by Basic.

Finally, we have three methods of embedding machine code programs in Basic by:

1. Creating space at the top of memory by moving the Basic TOM pointers down and using the space so reserved to load our machine code bytes from DATA statements. If the memory size is variable (i.e. you want the program to run on different VZ machines) then the machine code cannot contain any absolute memory references to within the program itself. You have a large area available for your code using this method.

If, however, you are writing the code for a fixed memory size, then of course the code can be assembled for that fixed memory location. You can choose a fixed top of memory value which will work for different VZ configurations, a simple method but wasteful of available memory space.

2. For small programs (approx. 60 bytes long) you can load your code into a REM statement in the very first line of your Basic program. This code can contain absolute memory references because the start of Basic is known for all memory sizes, but it cannot contain any zero bytes.

3. Raising the 'End of Basic' pointer value reserves space between the real end of Basic text and the variable storage area. Requires keeping track of where M/C program is being loaded to ensure that Basic text and M/C program code do not overlap.

7.16 LOADING FROM DISK

Another variation on the method of loading machine code from a Basic program exists, but is only convenient for programs loaded from Disk. In this method the required memory is reserved by moving the start of the Basic program area upward in memory from its normal start at 31465 (7AE9H).

Before you start typing in the Basic program proper, the 'Start of Basic program' and 'Start of Variables Table' pointers must be altered to the higher position. Now any Basic program lines entered will be placed into memory starting at this higher position. Your machine code program can now be loaded between the old start of Basic (31465) and the the new, higher start of Basic.

For tape users this method has one disadvantage. Before the main Basic program is reloaded, the 'Start of Basic program' and the 'Start of Variables Table' pointers will have to be altered by the 'shifting' program, otherwise the Basic Interpreter will not find the up-shifted program lines (i. e. it will not LIST or RUN).

This is because while the tape loading routine will place the Basic program into the same higher location from which it was CSAVEd, it does not reset the Basic program pointers to the new, higher position.

However, Disk System users will find that once the up-shifted Basic program has been entered and SAVEd on Disk, loading in the program from Disk will automatically reset the pointers to the the correct memory position, and consequently, the up-shifted Basic program will both LIST and RUN. The short Basic program in Listing 7.6 will reset the pointers to the required position.

Listing 7.6

```
100 INPUT"NEW START OF BASIC";NS
110 FOR J=NS TO NS+2
120 POKE J,0
130 NEXT J
140 M1=INT(NS/256):L1=NS-(M1*256)
150 POKE 30884,L1:POKE 30885,M1
160 M2=INT((NS+2)/256):L2=(NS+2)-(M2*256)
170 POKE 30969,L2:POKE 30970,M2
```

Lines 110 - 130 place three zero bytes into the new start of Basic area, while lines 140 - 170 calculate the 8-bit bytes to set the 'Start of Basic' and 'Start of Variable Table' pointers.

Incidently,once you have RUN this program, you will have lost it, because it no longer starts at the new 'Start of Basic' location. So, before RUNning it be sure to save it !

Once the program has been RUN, you can begin typing in your main Basic program, from which, presumably, you will load your machine code program from DATA statements into the area so reserved.

Remember, if you are using tape saving you must load and RUN the above 'shifting' program before attempting to reload/load your main program from tape. Disk users need not bother with this, just reloading the up-shifted program from Disk will reset the Basic pointers to the correct values.

Finally, the machine code program that you load below the up-shifted Basic program can contain absolute memory references because the start of the machine code program is fixed (31465) irrespective of the memory size of the machine used. In this respect, this method has the advantage of the REM method without the restriction of the limited program length allowable. In fact, the length of the machine code program is limited only by the size of the total memory available and the need to accommodate the main Basic program above it (between the machine code program and TOM).

-ooOoo-

NOTES

CHAPTER 8

ARITHMETIC OPERATIONS

8.Ø1 NUMBERING SYSTEMS

Before we can progress any further in our understanding of the operation of the VZ and the inner workings of the Z8Ø microprocessor chip, we need to study the way in which numbers are handled by the computer.

To start with, it is useful to have a closer look at the numbering system we use every day. This system is so familiar to us that we use it without giving a thought to the scheme of things behind the rules that we automatically use every day. Of course, as we know there are only ten different symbols in our decimal numbering system. Also obviously, we regularly use these symbols to represent quantities much larger than ten.

The method we use to do this is to simply create columns of symbols with the different columns having different weights assigned to their symbols. When we count using this system, we start from Ø, then progress through 1, 2, 3, 4, 5, 6, 7, 8, 9 (our ten symbols).

At this point we have run out of symbols. Our next logical step is to start back at Ø again and continue counting. However, we need some way to record the count of ten we have already completed. We do this by creating another column to the left of our first column. To indicate we have completed one count of ten we place the symbol 1 in this second column. As the count continues, we are going to eventually run out of symbols in the second column as well. We then simply create a third column to left of the second, and increment it by one symbol every time the second column **overflows**.

This process can be continued as long as we like to represent larger and larger numbers (provided we have a piece of paper large enough to write down all our columns of symbols).

So then, when we count in our decimal system, we simply increment through the list of ten symbols available to us. When we run out of symbols we create another column to indicate the overflow. The process just described is the same for any numbering system, whether it be HEXADECIMAL (six plus ten = sixteen different symbols), OCTAL (eight different symbols), or BINARY (two different symbols).

One final point about the decimal numbering system, if we look more closely at the effect symbols have on the actual quantity, we see that the symbols in each column have ten times the weight of the symbols in the column to its right. Also, the actual value of each symbol in a column is the symbol value (0-9) times the weight for that column.

Thus the number 5482 is evaluated as:-

DECIMAL NUMBER	:	5		4		8		2
COLUMN WEIGHT	:	10*10*10		10*10		10		1
COLUMN VALUE	:	5*1000		4*100		8*10		2*1
TOTAL VALUE	:	5000	+	400	+	80	+	2

= 5482

8.02 BINARY NUMBERS

The two most common numbering systems (other than decimal) that you will encounter are BINARY and HEXADECIMAL. We will now look at the simpler of the two systems, BINARY.

As mentioned previously, the binary numbering system has only two symbols. To count in binary we follow the same procedure as for our familiar decimal system. However, because we have only two symbols in our binary system, we very quickly run out of symbols. This means that we have to start creating those extra columns at a rapid rate.

Hence, a quantity written out in binary notation is much longer than the same quantity written in decimal notation. Additionally, because we run out of symbols after two increments, the weight for each column is **TWO** times the weight of the column to its right.

That is, the right-most bit (BI-nary digi-T) in a binary number has the smallest weight and so is called the Least Significant Bit (LSB), while the left-most bit has the largest weight and so is referred to as the Most Significant Bit (MSB). Thus the binary number 11101 is evaluated in decimal as:-

BINARY NUMBER:	1		1		1	0		1
COLUMN WEIGHT:	2*2*2*2		2*2*2		2*2	2		1
COLUMN VALUE :	1*16		1*8		1*4	0*2		1*1
TOTAL VALUE :	16	+	8	+	4	+ 0	+	1

= 29 decimal

One of the biggest surprises to newcomers to M/C programming is the realisation that although M/C allows you to do many things that Basic will not allow, it also demands that even the simplest operation must be done at the lowest level.

For example, in Basic the programmer would barely blink an eye at adding or multiplying two numbers, while in M/C these simple operations can be quite complex to program. This is especially true when you are dealing with floating point numbers (numbers which contain decimal points e.g. 77.352).

Specialised routines need to be developed to handle these floating point numbers. Once again, it would be easier to use Basic to do these arithmetic operations.

In this chapter then, we will deal with simple arithmetic operations only.

However, once again, it is stressed that using Basic code is by far the simplest method of performing arithmetic operations in your program.

8.Ø3 EIGHT-BIT ARITHMETIC

In arithmetic operations the most important register is the Flag or 'F' register. This register is different from any of the other registers in the Z8Ø CPU.

You cannot perform any data transfer operations, in fact, the only direct operations that can be carried out on the flag register are PUSH-ing and POP-ping to the stack and clearing the carry bit.

The state of the bits in the flag register is determined by the results of operations carried out in the A register and cannot be altered directly except for the carry bit.

The contents of the flag register should not be considered to be a byte, but rather a collection of bits used to 'flag' various conditions that have occurred. Only 6 out of the 8 bits are used, these being SET ('1') or CLEARED ('Ø') depending on the outcome of some operation.

Not all operations perform by the CPU affect the condition of the Flag bits, for example, load instructions do not affect any.

The most important flag bit in the flag register is the CARRY flag. Remembering that the maximum number that can be stored in an eight bit number is 255, what happens when we add two numbers in the accumulator (the A register) which total more than 255 ?

For example, consider the addition of 255 plus 1 = 256 :-

```
   11111111 (255)
+  00000001   (1)
  100000000 (256)
```

The result (256) requires nine bits, but the registers and memory devices are only eight bits in an eight-bit computer. If we ignore the ninth bit, we will have the result 255 + 1 = 0. One purpose of the carry bit in the flag register is to signal the occurrence of such an overflow (ninth bit) resulting from an addition.

An overflow condition also occurs when we perform a subtraction which results in a negative answer. The carry bit is set to signal that a borrow has occurred.

The particular condition of the carry flag can be used directly in the arithmetic operation or can be used to control the path of execution of the program.

To illustrate the use of the carry flag bit, we will look at the programming required to perform simple integer addition operations.

8.04 ADDING TWO NUMBERS

Machine programming is largely a matter of moving data into areas where it can be accessed by the CPU to allow operations to be carried out on that data. That is, the main area of activity is storing and retrieving data.

This is why LOAD instructions form such a large part of the CPU instruction set. LOAD instructions are used to move the data around inside the microprocessor system into areas where operations can be carried out (mainly in registers) and then stored back in memory in known locations for further processing or for later use elsewhere in the program.

The following programs compare the operations needed to add two numbers between 0 and 255, the first in Basic and the second using M/C.

Listing 8.1

```
100 INPUT A                 :REM PUT FIRST NUMBER INTO 'A'
200 INPUT B                 :REM PUT SECUND NUMBER INTO 'B'
300 R = A + B               :REM ADD TWO NUMBERS, RESULT IN 'R'
400 PRINT R                 :REM PRINT RESULT ON SCREEN
500 GOTO 100                :REM REPEAT
```

The short Basic program in Listing 8.1 accepts two numbers, adds them, and stores the result in the variable 'R'.

The program in Listing 8.4 uses Basic to input the two numbers to be added, but instead of loading them into variables 'A' and 'B', it transfers the two numbers to two consecutive memory locations (8000H and 8001H).

A jump is made to the M/C routine, where the first number is loaded into the B register from location 8000H, the second number into the A register from location 8001H, then the two numbers are added together by the CPU.

The CPU leaves the result in the A register from where it is loaded back into a third memory location (8002H).

A return is made back to Basic where the result in loaded from memory into the variable 'R' and then displayed.

Firstly, the M/C source code:-

Listing 8.2

```
001 ; ADDS TWO 8-BIT NUMBERS
002 ; STORED IN 8000H & 8001H
003 ; RESULT IS LEFT IN 8002H
004 ADD1 DEFB 00H
005 ADD2 DEFB 00H
006 RES1 DEFB 00H
007 STRT LD   A,(ADD1)
008      LD   B,A
009      LD   A,(ADD2)
010      ADD  A,B
011      LD   (RES1),A
012      RET
```

In the source code in Listing 8.2 we have used the DEFB (DEFine Byte) pseudo-op to reserve three consecutive bytes and label them ADD1, ADD2 and RES1 respectively.

Line 007 instructs the CPU to get the byte stored in location ADD1 into the A register.

Line 008 then transfers that number into the B register. This is because the B register cannot be loaded directly from memory as can the A register.

Line 009 tells the CPU to load the second number from location ADD2 into the A register.

Line 010 instructs the CPU to add the contents of the B register to the A register and store the result back into the A register. Note that this results in the original contents of the A register being lost (or more correctly overwritten).

Line 011 loads the result in the A register into location RES1 where it will be accessed after the return to Basic is made in Line 012.

The machine code is assembled and loaded at 8000H. This is done without the usual reserving of memory for the machine code. This can be 'got away with' because both the machine code and the Basic program that goes with it are small.

We can load the machine code in the 'no-man's land' between the end of the Basic program itself and the Basic top-of-memory boundary without fear of it being overwritten.

When assembled starting at location 8000H, by hand or by the Editor Assembler, the source code in Listing 8.2 produces the object code in Listing 8.3 (Refer Appendix 6 for object codes).

Listing 8.3

ADDRESS	OBJECT CODE
8000	00
8001	00
8002	00
8003	3A 00 80
8006	47
8007	3A 01 80
800A	80
800B	32 02 80
800E	C9

Note here that the first three bytes are reserved by the DEFB pseudo-op for the storage of the two numbers to be added and storage of the result. That is, the actual start of executeable code is at 8003H. This is the location that must be entered into the USR function execute address pointer at 788EH and 788FH (30862 and 30863 decimal).

The Basic program used to load and call the above M/C program is given below. Note once again how the USR execute address (8003H) has been poked into 30862 and 30863. Also note line 50 contains the decimal equivalents of the hexadecimal object code for the routine as given above, which is poked into memory by lines 10 to 40 of the Basic loader.

Listing 8.4

```
10 FOR I = 0 TO 14                    :REM POKE THE ROUTINE INTO
20 READ D                             :REM MEMORY, STARTING AT 8000H
30 POKE -32768+I,D
40 NEXT I
50 DATA 0,0,0,58,0,128,71,58,1,128,128,80,2,128,201: REM CODE
60 POKE 32867,3:POKE 32868,128:REM SET UP USR ADDRESS (8003H)
100 INPUT A                           :REM PUT FIRST NUMBER INTO 'A'
200 INPUT B                           :REM PUT SECOND NUMBER INTO 'B'
300 POKE -32768,A                     :REM FIRST NUMBER INTO 8000H
310 POKE -32767,B                     :REM SECOND NUMBER INTO 8001H
320 X = USR(0)                        :REM CALL ADD ROUTINE
330 R = PEEK(-32766)                  :REM GET RESULT FROM 8002H
400 PRINT R                           :REM PRINT RESULT ON SCREEN
500 GOTO 100                          :REM REPEAT
```

Of course you can input any size number into the pure Basic only program as the interpreter will automatically account for any result. If the result is larger than 6 digits then it will use exponential notation to express the answer.

When you run the M/C add program, you will find some curious results. Input '200' for both the first and second number. You would expect the result to be 400 (quite rightly, as 200 + 200 = 400). However, the M/C add program returns not 400, but 114.

This seems odd until it is remembered that the maximum number the A register can hold is the maximum value of an eight-bit number. This maximum number, you should recall, is 255.

To represent the result 400 we need nine bits, this ninth bit would have a weight of 256. However,the A register is only eight bits wide, therefore when the addition of two numbers in the A register exceeds 255, the ninth bit value of 256 is lost from the result.

We can check this by subtracting 256 from the expected result (400):-

400 - 256 = 114

This is the result that is returned by our M/C add program. Obviously, this is not a very satisfactory state of affairs, so we need some way of retaining this overflow. This is done by a special bit reserved in the Flag register. When we add two numbers in our familiar decimal system which total more than 9, we keep track of the overflow by 'carrying' to the next column to the left. The special bit in the flag register which imitates this operation is called the Carry bit.

We have to modify our M/C routine to test for the carry caused by the result being greater than 255. We could use a fourth location to 'flag' whether there has been a carry or not. We could load this carry location with a '1' if there is a carry, a '0' if there is not.

Listing 8.5

```
001 ; ADDS TWO 8-BIT NUMBERS
002 ; STORED IN 8000H & 8001H
003 ; RESULT IS LEFT IN 8002H
004 ; CARRY FLAG IN 8003H
005 ADD1 DEFB 00H
006 ADD2 DEFB 00H
007 RES1 DEFB 00H
008 CARR DEFB 00H
009 STRT LD   A,(ADD1)
010      LD   B,A
011      LD   A,(ADD2)
012      ADD  A,B
013      LD   (RES1),A
014      LD   A,00H
015      JR   NC,ZERO
016      INC  A
017 ZERO LD   (CARR),A
018      RET
```

Notice how in Line 008 an extra byte has been reserved for the carry flag (8003H). A test for the condition of the carry flag is made in Line 015, and if the carry flag is not set (No Carry is true), then a relative jump is made to the label 'ZERO' at Line 017. Here the zero byte which was loaded into the A register at Line 014 is loaded into CARR, our carry flag location 8003H.

If a carry did occur, then the increment instruction of Line 016 is not skipped and the zero byte in the A register is incremented to a '1'. This value is then loaded into the CARR location to signal that a carry has been made in the addition.

Note that the loading of a zero byte into the A register in Line 014 did not affect the carry flag, thereby allowing a test of the previous addition to be carried out in Line 015.

The Basic loader for the M/C program will have to be modified to include the extra code as well as the additional check for the carry flag. In particular note that the start of executeable code has been shifted up by one location (to 8004H) to make way for the carry flag byte at 8003H.

Listing 8.6

ADDRESS	OBJECT CODE	ADDRESS	OBJECT CODE
8000	00	800B	80
8001	00	800C	32 02 80
8002	00	800F	3E 00
8003	00	8011	30 01
8004	3A 00 80	8013	3C
8007	47	8014	32 03 80
8008	3A 01 80	8017	C9

Listing 8.7

```
10 FOR I = 0 TO 23                    :REM POKE THE ROUTINE INTO
20 READ D                             :REM MEMORY, STARTING AT 8000H
30 POKE -32768+I,D
40 NEXT I
50 DATA 0,0,0,0,58,0,128,71,58,1,128,128,50,2,128:REM CODE
55 DATA 62,0,48,1,60,50,3,128,201                 :REM CODE
60 POKE 30862,4:POKE 30863,128:REM SET UP USR ADDRESS (8004H)
100 INPUT A                           :REM PUT FIRST NUMBER INTO 'A'
200 INPUT B                           :REM PUT SECOND NUMBER INTO 'B'
300 POKE -32768,A                     :REM FIRST NUMBER INTO 8000H
310 POKE -32767,B                     :REM SECOND NUMBER INTO 8001H
320 X = USR(0)                        :REM CALL ADD ROUTINE
330 R = PEEK(-32766)                  :REM GET RESULT FROM 8002H
340 IF PEEK(-32765) = 1 THEN R = R + 256:REM ADJUST IF CARRY
400 PRINT R                           :REM PRINT RESULT ON SCREEN
500 GOTO 100                          :REM REPEAT
```

8.05 ADDING LARGER NUMBERS

When you run the previous M/C add program, you will find that entering 200 for the two numbers will now produce the correct answer of 400. But you will find entering numbers greater than 255 for either of the two input numbers will produce a **'FUNCTION CODE ERROR'**.

This is because you have attempted to use the POKE function with a number which is larger than the maximum number that a single memory location can store. Remember each memory location can only store one eight-bit byte with a maximum value of 255.

When doing arithmetic there obviously is a need to deal with numbers larger than 255. Numbers over 255 can be stored in two bytes, called the 'high order byte' and the 'low order byte', just like addresses.

The high order byte contains the number of whole 256s there is in the number, while the low order byte stores the fractional part of 256 which makes up the remainder of the number.

Remember, as with addresses, the Z80 CPU always accesses the low order byte before the high order byte and so you must ensure that you store them in that order in memory.

Just like addresses, to express a number in two bytes, first find how many 'whole lots of 256' there are in the number. That is, divide the number by 256. The integer part of the answer is the decimal equivalent of the high order byte. The decimal fraction part of the answer is the fraction of 256 which makes up the remainder. To convert this decimal fraction to a whole number between 0 and 255, multiply by 256.

For example, the number 1574:-

1574 / 256 = 6.1484

Therefore, the decimal equivalent of the high order byte is 6, and the decimal equivalent of the low order byte is 0.1484 * 256 = 37.99 or 38. You can avoid the error in the low order byte by finding the remainder by subtracting the whole lots of 256 from the original number, the answer is the remainder.

1574 - (6 * 256) = 38

Hence, the number 1574 is stored in memory as two bytes, the first containing 38 decimal (LSB), the second containing 6 decimal (MSB).

We now need a total of seven bytes to perform the addition of two 2-byte numbers. Two for the high and low bytes of the first number to be added, two for the second number, two for the result, and one to store the carry from the addition, if any.

Using the accumulator to do the additions piecemeal is tedious because the A register can only handle one byte at a time. Fortunately, the HL and BC register pairs can be used as limited 16-bit (or 2-byte) accumulators. Actually the HL register is the accumulator, while the BC register can be used to store the other half of the 2-byte addition.

Listing 8.8 is the source and object code for a 16-bit addition subroutine which will be loaded and called as in the previous examples. The origin is set to 8000H again.

Note the use of the DEFW (DEFine Word) pseudo-op to reserve two consecutive bytes for each 16-bit quantity. (See Chapter 11).

After the addition in Line 011 the answer is left in the HL register pair. If the addition results in a carry, the carry bit of the flag register is set. A different method is used here to test the condition of the carry bit (Lines 013 & 014) and set the contents of CARR (8006H) accordingly. If you use the instruction ADC instead of ADD, the condition of the carry bit is added to the result.

Therefore, if you load the A register with zero, and then add it to zero again, the result will be the condition of the carry bit.

That is, if the carry bit was set, then after the ADC instruction has been executed the A register will contain a '1'. If the carry bit was cleared (zero-ed), then the A register will contain zero. The resultant contents of the A register are then loaded into CARR for later access from Basic via a PEEK.

Listing 8.8

```
001 ; ADDS TWO 16-BIT NUMBERS
002 ; IN 8000/1H & 8002/3H
003 ; RESULT IN 8004/5H
004 ; CARRY IN 8006H
005 ADD1  DEFW 0000H
006 ADD2  DEFW 0000H
007 RES1  DEFW 0000H
008 CARR  DEFB 00H
009 STRT  LD   BC,(ADD1)
010       LD   HL,(ADD2)
011       ADD  HL,BC
012       LD   (RES1),HL
013       LD   A,00H
014       ADC  A,00H
015       LD   (CARR),A
016       RET
```

```
8000 00 00
8002 00 00
8004 00 00
8006 00
8007 ED 4B 00 80
800B 2A 02 80
800E 09
800F 22 04 80
8012 3E 00
8014 CE 00
8016 32 06 80
8019 C9
```

The Basic loader is given in Listing 8.9 for the 16-bit M/C add program.

The M/C 16-bit add program loaded from Basic in Listing 8.9 will accept input numbers from 0 to 65536.

This is still a long way from the pure Basic only addition program given at the beginning of the chapter, which will accept floating point numbers in the range from -1.70141E+38 to 1.70141E+38.

Listing 8.9

```
10 FOR I = 0 TO 26                    :REM POKE THE ROUTINE INTO
20 READ D                             :REM MEMORY, STARTING AT 8000H
30 POKE -32768+I,D
40 NEXT I
50 DATA 0,0,0,0,0,0,0,237,75,0,128,42,2,128        :REM CODE
55 DATA 9,34,4,128,62,0,206,0,50,6,128,201         :REM CODE
60 POKE 30862,7:POKE 30863,128:REM SET UP USR ADDRESS (8007H)
100 INPUT A                           :REM PUT FIRST NUMBER INTO 'A'
200 INPUT B                           :REM PUT SECOND NUMBER INTO 'B'
210 MA = INT(A/256):LA = A - (MA*256)    :REM 'A' INTO 2 BYTES
220 MB = INT(B/256):LB = B - (MB*256)    :REM 'B' INTO 2 BYTES
300 POKE -32768,LA                    :REM PUT LSB OF 'A' INTO 8000H
305 POKE -32767,MA                    :REM PUT MSB OF 'A' INTO 8001H
310 POKE -32766,LB                    :REM PUT LSB OF 'B' INTO 8002H
315 POKE -32765,MB                    :REM PUT MSB OF 'B' INTO 8003H
320 X = USR(0)                        :REM CALL ADD ROUTINE
330 LR = PEEK(-32764)                 :REM LSB OF RESULT FROM 8004H
335 MR = PEEK(-32763)                 :REM MSB OF RESULT FROM 8005H
338 RR = LR + (256 * MR)              :REM TWO BYTES INTO RESULT
340 IF PEEK(-32762) = 1 THEN RR = RR + 65536  :REM IF CARRY
400 PRINT RR                          :REM PRINT RESULT ON SCREEN
500 GOTO 100                          :REM REPEAT
```

Writing routines to handle floating point numbers is well beyond the scope of this book and such operations can be handled by converting the floating point data into integers and then use routines which are extensions of the above.

8.06 MANIPULATING BINARY NUMBERS

We have just seen how we can add two numbers via machine code, however, we did not actually see how the CPU performs these additions on those numbers. To be able to manipulate binary data in a more general way we need to look at the way the CPU handles the numbers presented to it.

We already know that the computer only deals in binary data of eight bits, with each 8-bit byte occupying one memory location. These 8-bit bytes can represent directly the numbers between 0 and 255.

Of course we need to handle other data than numbers between 0 and 255 in our computer, and we have looked at how several consecutive bytes in memory can be used to represent numbers larger than 255 in the previous examples. However, the computer must have some means to represent the other forms of information that it is required to handle.

In Basic we use several types of variables in our programs to hold the information that we want the program to work on. These are:-

- real : can have a decimal fraction, e.g. A = 3.56, B = 1.6E+32

- integer : cannot have a decimal fraction, e.g. C% = 122

- string : used to store a string, e.g. D$ = "TEST"

When we use machine code, we cannot use these variables directly, but we have to write software which will represent and handle this data using only integers in the range Ø to 255. We can combine bytes in a fixed format to represent larger numbers or floating point (real) quantities.

The CPU does not inherently 'know' what data any combination of 8-bit bytes represent, that 'knowledge' is contained in the manner in which the data is manipulated by software that must be written.

8.Ø7 SIGNED INTEGERS

We have seen how the CPU handles integers in the range of Ø to 65535, but what about negative numbers ? Negative numbers must still be represented by binary numbers so that they can be handled by the CPU, but we need some way of indicating the sign of the number.

The most common way is to use the left-most bit in the number (the MSB) to indicate whether the number (represented by the remaining bits to the right) is negative or positive. To signal that the number is negative, the MSB of the number is set to '1'. A positive number is signalled by the MSB being cleared to a 'Ø'.

The effect of using one bit to signal the sign of the number is to reduce (by one) the number of bits available to represent the actual quantity of the number. Because it is the MSB that is used, the range is reduced by a factor of two. The new range for an 8-bit number is -128 to +127, and for a 16-bit number the range is -32768 to +32767.

How do we (or the CPU for that matter) distinguish a large un-signed integer (Ø-65535) from a negative signed integer?

We cannot. Nor can the CPU.

It is up to the programmer to interpret the data when it is input and when it is output as a result.

The manipulation of signed integers is one area which causes a certain degree of heartburn for beginner programmers.

We need to determine the actual code that is used to represent the negative quantity. First, we make the assumption that the 8-bit binary number '00000000' represents the number zero. Now of course, the addition of a negative and a positive number of the same magnitude must equal zero, e.g. (+1) + (-1) = 0.

Therefore:-

```
        00000001   (+1)
    +   xxxxxxxx   (-1)
        00000000    (0)
```

Now the rules of binary addition are very simple:-

```
0 + 0 = 0
0 + 1 = 1
1 + 1 = 0 plus carry of '1' to the column to the left.
```

To get the LSB (right-most bit) of the result of the addition to equal '0', the LSB of the binary representation of -1 must be '1' as 1 + 1 = 0.

The carry from the addition in the LSB column is moved to the left. Now, to get the result in the second column to be '0', the second bit of -1 must also be '1', and so on.

The representation of -1 in binary is therefore:-

```
xxxxxxxx = 11111111
```

Repeating for -5 :-

```
        00000101   (+5)
    +   xxxxxxxx   (-5)
        00000000    (0)
```

Using the same rules, the bit pattern as follows will produce zero when added to +5.

```
xxxxxxxx = 11111011
```

You are probably wondering at this stage if there is an easier way than this try-and-see method!

Fortunately, there is. To find the binary representation of a negative number, take each bit of the binary pattern for the positive value of the number and invert. This is called complementing the number.

The result of the inversion of each bit of the positive value is called the ONE's COMPLEMENT. If we now add 1 to the one's complement, we have the TWO's COMPLEMENT. This two's complement is the representation for the negative number. That is, for the number -5 we find the binary representation by:-

```
    00000101   (+5)
    11111010   (complementing each bit of +5)
+   00000001   (add one to the one's complement)
    11111011   (two complement result = -5)
```

This technique can be extended to 16-bit numbers to give the range of -32768 to +32767. This is the range of legal numbers that can be used in the Basic memory commands, PEEK and POKE.

8.08 STRINGS (OF CHARACTERS)

When we 'string' together characters into a group, we can form them into words, then into text. There are many machine code programs that are used to handle text in the computer (e.g. word processor programs), but so far we have only seen how the binary patterns inside a computer can be used to represent numbers.

So, in order to handle text characters inside the computer, we encode them. There is a standard code used for text characters, called the American Standard Code for Information Interchange, thankfully shortened to the ASCII code.

For example, as mentioned earlier, the code for the letter 'A' is 65 decimal (41H). You can find the ASCII code for any character (including the 'RETURN' key) by using the following Basic program.

Listing 8.10

```
100 I$ = INKEY$:I$ = INKEY$
200 IF I$ = "" THEN 100
300 R$ = INKEY$:R$ = INKEY$
400 IF R$ <> "" THEN 300
500 PRINT"THE ASCII CODE FOR";T$;"IS";ASC(T$)
600 GOTO 100
```

The first character to be output will be the 'RETURN' which is typed to 'RUN' the program itself. After that the program will return the ASCII code for any character that can be typed from the keyboard, including the graphics characters (for the colour green).

Therefore, a binary pattern in memory can represent the following information:-

1. A machine code instruction which can be executed directly by the CPU.

2. A numerical value within the range 0 to 255, or by grouping two bytes together, within the range 0-65535.

3. A code representing a character.

When the CPU encounters these bytes in memory, it cannot distinguish between them, it is up to the programmer to instruct the CPU to treat the binary information in the correct manner by the structure of the program and by the manner in which the data is used.

-ooOoo-

CHAPTER 9

THE VIDEO SCREEN - MESSAGES & SIMPLE GRAPHICS

In this chapter we will apply some of the techniques already discussed in previous chapters by writing a small M/C program which will output a short message on the screen. To run the program the message is first loaded by Basic into a specific area which we will call the text buffer. Then Basic calls (via a USR function call) the M/C program which takes each letter of the message and outputs it to the screen.

9.01 END OF MESSAGE FLAG

We need some way of signalling to the M/C program when it has reached the end of the message data because we want to be able to enter messages of different lengths. If we are only outputting text then we can use one of the graphic block codes as a flag byte for the end of the message. For example, we might choose 255 (0FFH) as the end of message marker, i.e. 0FFH is loaded to the end of the text message in the text buffer.

As the program is reading the letter codes from the text buffer, it checks each code to see if it is 0FFH. If the code equals 0FFH then the program returns control back to Basic, otherwise the code is a valid character code and is output to the screen.

9.02 FINDING THE END OF MESSAGE FLAG

To detect whether the code is 0FFH we use a comparison instruction to compare the code we have loaded into the A register from the text buffer to 0FFH. The 'CP' instruction actually does a non-destructive subtraction of the value specified in the second byte of the instruction and the value held in the A register.

If the answer is zero then the two values must be equal and this sets the zero flag in the flag register to 1.

The next instruction after the compare operation is a conditional return instruction which tests the state of the zero flag, and if it is set then the last code compared must have been 0FFH, so a return is made to Basic.

9.03 SPACE FOR SMALL TEST PROGRAMS

Memory space is needed not only for the text buffer, but also the program itself. Because the program is small there is no need to lower the top of memory pointers to reserve space, but instead the program can be loaded mid-memory and still be out of the way. As the text buffer will be variable in length, it makes good sense to put this after the M/C program code proper.

For small experimental programs I use 8000H as a starting point for the M/C program itself and 8800H for the start of any data areas. This leaves plenty of space for the small Basic loader program below it and more than enough space from the top of memory for working space for the Basic interpreter to use.

Our text buffer begins from 8800H, but the start of the video display area must also be known. Chapter 7 gave that as 7000H (28672 decimal), so if we wanted to print the message on the top line then we would use this value.

Of course, if the message needs to be displayed further down the screen, we would use a starting address higher in memory - as long as the text is not loaded higher than 71FFH which is the bottom right-hand screen position in MODE(0).

9.04 THE MESSAGE PROGRAM

Listing 9.1 and 9.2 are the mnemonics and hex codes respectively for the message output program. A slight complication here is that an absolute jump is made in the last line (line 009) as part of the looping process to get the next letter code. This means we have to find the absolute memory address of the third instruction in the program and enter it into the last instruction.

NOTE - comments in brackets in listings in this chapter are for explanation only and are not to be typed into the program listing.

Listing 9.1 Mnemonics

```
001     START   LD      BC,8800H        (start of text buffer)
002             LD      DE,7000H        (start of video display)
003     LOOP    LD      A,(BC)          (get next letter code)
004             CP      0FFH            (compare code with 0FFH)
005             RET     Z               (back to Basic if 0FFH)
006             LD      (DE),A          (load code to screen)
007             INC     BC              (adjust BC to next code)
008             INC     DE              (DE to next screen posn.)
009             JP      LOOP            (go back for next code)
```

Listing 9.2 Addresses and Hex Codes

```
001     8000    01 00 80          (note reverse order
002     8003    11 00 70            of address bytes)
003     8006    0A          (back to this address from last line)
004     8007    FE FF         (second byte is the compare value)
005     8009    C8             (go back to Basic if code = 0FFH)
006     800A    12                   (must be valid letter code)
007     800B    03                 (next code from text buffer)
008     800C    13               (next screen location for code)
009     800D    C3 06 80      (jump back to 8006H, note reverse
                                    order of address bytes)
```

In lines 003 and 006 the operands are in brackets, being specified by the current contents of the register pairs, BC and DE. The operands are in effect variable operands because the actual value can be changed by a simple increment instruction operating on the register pairs as in lines 007 and 008.

This form of variable addressing is called 'indirect addressing' because the CPU has to first read the instruction code in memory, then go to the register pair specified and then pull out the actual address to be used.

In lines 001, 002 and 009 the addresses are actually written into the instruction, so the CPU can read the address required directly from the code in program memory. This type of direct reading of the required address is called 'immediate addressing'.

The first two lines initialise the two register pairs to the start of the text buffer (BC) and the start of screen memory (DE).

Line 003 instructs the CPU to read the address value from the BC register pair and then put the byte value stored at this address into the A register. This is indirect addressing as described above.

Line 004 compares the byte just loaded into the A register with 0FFH and sets the zero flag bit in the flag register to 1 if the byte in the A register is 0FFH.

Line 005 returns control to Basic if the test for 0FFH is true, ie. the zero flag bit is set.

Line 006 is executed if the the byte tested in line 004 is not 0FFH, that is, a valid letter code. The valid letter code in the A register is loaded into the current address value held in the DE register pair.

Lines 007 and 008 adjust the register pairs BC and DE, to their respective next addresses ready for the processing of the next letter code byte.

Line 009 causes an unconditional jump back to line 003 where the next code is loaded from the text buffer and the process continues.

Listing 9.3 Basic Loader

```
100 ' HEX LOADER PROGRAM
110 SP=32768                          (start of M/C program - 8000H)
120 READ HH$                         (read the next hex data string)
130 IF HH$="XX" THEN  250              (test for end of data flag)
140 IF LEN(HH$)<>2 THEN  240             (valid hex code length)
150 N=0
160 FOR I=1 TO 2                       (two digits to be processed)
170   V=ASC(MID$(HH$,I,1))              (ASCII code of each digit)
180   IF (V<48)OR(V>70)OR((V>57)AND(V<65)) THEN 240
190   N=N*16+(V-55+(-7*(V<64)))          ( build decimal value)
200 NEXT I                                  (get second digit)
210 POKE (SP+(65536*(SP>32767))),N    (load value into memory)
220 SP=SP+1                        (adjust pointer to next location)
230 GOTO 120                                (do next hex digit)
240 PRINT"ERROR IN HEX DATA":END                 (error message)
250 MS=INT(SP/256)                 (find page byte - MSB of program)
260 LS=SP-(MS*256)                      (start and then find LSB)
270 POKE 30862,LS:POKE 30863,MS      (poke LSB & MSB into USR)
280 ' TX$ IS THE TEXT TO BE OUTPUT
290 INPUT"TYPE IN TEXT TO BE OUTPUT";TX$          (input text)
300 SB=34816                          start of text buffer - 8800H)
310 FOR I=1 TO LEN(TX$)                  (go for length of text)
320   LC=ASC(MID$(TX$,I,1))          (find ASCII value of letter)
330   POKE (SB+(65536*(SB>32767))),LC     (letter into buffer)
340   SB=SB+1                          (adjust text buffer pointer)
350 NEXT I                                  (go to last letter)
360 POKE (SB+(65536*(SB>32767))),255   (end buffer with 0FFH)
370 X=USR(0)                                (go to M/C program)
380 PRINT@256,"FINISHED":END              (return here and end)
380 ' HEX CODES
390 DATA 01,00,88,11,00,70,0A,FE,FF,C8,12,03,13,C3,06,80
400 DATA XX
```

9.05 THE BASIC LOADER

The loader program in Listing 9.3 is slightly different to previous versions as the hex codes are entered into DATA statements of the loader without first converting them to their decimal equivalents. This has the advantage of being less prone to errors, but is slower during the loading process.

9.06 GRAPHICS PROGRAM

This program will fill the screen with any character specified including both letters and graphics characters. Of course, much more complex programs have to be written for something spectacular, but with the simple program here you can concentrate on fundamental principles.

Listing 9.4 and 9.5 contain the mnemonics and the object codes for the screen fill program respectively.

Listing 9.4 Screen Fill Program Mnemonics (Source Listing)

```
001     START   LD      HL,7000H        (start of video display)
002             LD      BC,0200H    (number of screen locations)
003     LOOP    LD      A,(7921H)    (get passed character code)
004             LD      (HL),A          (load to screen memory)
005             INC     HL               (next screen position)
006             DEC     BC             (decrement byte counter)
007             LD      A,B              (load A with B register)
008             OR      C                   (OR with C register)
009             JR      NZ,LOOP          (do till end of screen)
010             RET                   (all done, back to Basic)
```

Listing 9.5 Screen Fill Program Object Code

```
001     8000    21 00 70  (note reverse order of address bytes)
002     8003    01 00 02        (512 screen locations - 0200H)
003     8006    3A 21 79         (code passed to 7921H by USR)
004     8009    77                  (load character to screen)
005     800A    23                       (next screen position)
006     800B    0B                 (keep track of bytes loaded)
007     800C    78          (get contents of B into A register)
008     800D    B1        (OR with C, if both B and C registers
009     800E    20 F6              are zero then zero flag set)
010     8010    C9                             (return to Basic)
```

Line 001 loads the start of the video display area into the HL register, while line 002 loads the BC register with the number of bytes required to be loaded. Here the whole screen is going to be filled which has 512 locations or bytes in memory, so 0200H is loaded into BC.

Line 003 is interesting because this is where the M/C retrieves data passed to it from the Basic USR function. When 'X=USR(CC)' is executed in the Basic program, the 16-bit (two-byte) value of CC is loaded to 7921H and 7922H (low byte first). Because the low order byte is loaded first in memory and because the number in CC should always be less than 255 (to be a valid character code), the A register needs only to be loaded with the contents of 7921H.

Line 003 must be executed each time a character is loaded because the contents of the A register is overwritten in line 006 during testing of the BC register.

The need to use registers for more than one task occurs frequently in M/C programming and care must be taken to keep certain values intact during program execution. Sometimes another unused register can be utilised or perhaps a temporary storage location within the program itself.

As a variation on this, the temporary storage area used here for the character code is outside the program area itself, in the USR function work area. However, the use of the stack is the most common way of temporarily storing vital data and is dealt with in more detail in the next chapter.

Line 004 loads the character code just received into the memory address pointed to by the HL register.

Line 005 increments the video memory pointer (HL register) to the next screen address while line 006 decrements the byte counter (BC register) one more count towards 0000H.

Lines 007 and 008 do a logical 'OR' of the low and high order bytes of the BC register. Only when both B and C registers contain zero will the zero flag bit be set in the flag register. The state of the zero flag bit is tested in line 009 and if not set (the contents of BC not equal to 0000H) then a jump is made back to line 003 to do the next byte.

Note how that while BC is loaded as register pair in line 002, each register can be manipulated separately, as in lines 007 and 008.

Notice also, the instruction in line 009 of the hex codes does not contain an absolute memory address reference to the address of the code in line 003, but has only a single byte, 0F6H. This is the value of the relative jump from the code in line 009 to the code in line 003 of the hex codes listing.

The use of relative jump addressing is very important because it allows the relocation of the code around in memory without changing the code. This is because the code contained in the instruction is not the actual memory address of the destination of the jump - which would have to be changed if the program was moved around - but the offset between the two sections of code which does not change.

The manual calculation of the offset is a bit tricky and will be shown in the next chapter on jumping and looping.

This is one area where you will appreciate the convenience of an Editor Assembler which calculates the offset automatically during assembly of the source listing. Line 010 contains the return to Basic when BC has been decremented to 0000H, i.e., all the screen has been filled.

Listing 9.6 contains the Basic loader for the screen fill program.

By the way, when the term "output to the screen" is used, it really is a bit misleading. The CPU does not output to the screen directly, but rather loads character codes into an area of RAM which has been connected to the Video Display Generator (VDG) chip by the hardware circuits.

It is the VDG which periodically reads this video RAM and generates the video signals which are fed to the screen via the video outputs producing the characters on the display.

Listing 9.6 Screen Fill Program Basic Loader

```
100 ' HEX LOADER PROGRAM
110 SP=32768                          (start of M/C program - 8000H)
120 READ HH$                          (read the next hex data string)
130 IF HH$="XX" THEN  250               (test for end of data flag)
140 IF LEN(HH$)<>2 THEN  240              (valid hex code length)
150 N=0
160 FOR I=1 TO 2                        (two digits to be processed)
170   V=ASC(MID$(HH$,I,1))                (ASCII code of each digit)
180   IF (V<48)OR(V>70)OR((V>57)AND(V<65)) THEN 240
190   N=N*16+(V-55+(-7*(V<64)))              ( build decimal value)
200 NEXT I                                      (get second digit)
210 POKE (SP+(65536*(SP>32767))),N    (load value into memory)
220 SP=SP+1                        (adjust pointer to next location)
230 GOTO 120                                  (do next hex digit)
240 PRINT"ERROR IN HEX DATA":END                 (error message)
250 MS=INT(SP/256)                 (find page byte - MSB of program)
260 LS=SP-(MS*256)                        (start and then find LSB)
270 POKE 30862,LS:POKE 30863,MS       (poke LSB & MSB into USR)
280 ' CC IS THE CHARACTER CODE TO BE OUTPUT
290 INPUT"TYPE IN CODE (0 - 255)";CC    (input character code)
300 X=USR(CC)                      (pass CC to location 7921H/7922H)
310 GOTO 310                                  (return here and end)
320 ' HEX CODES
330 DATA 21,00,70,01,00,02,3A,21,79,77,23,0B,78,B1,20,F6,C9
340 DATA XX
```

-ooOoo-

NOTES

CHAPTER 10

JUMPS, BRANCHES (AND MORE ON STACK OPERATIONS)

Many times while programming in Basic you need to jump to another section of the program, away from the straight execution of line numbers in sequence. The GOTO and GOSUB commands are used for this in Basic.

The GOTO command line can be either unconditional - 'GOTO 100', or conditional - 'IF X=1 THEN GOTO 100'.

In M/C programming there are also unconditional JUMPS and conditional BRANCHES to other parts of the program, with an extra twist as we will see. There is also the M/C equivalent to the GOSUB, a 'CALL' to a subroutine.

10.01 JUMPS, BRANCHES AND SUBROUTINES

During the normal uninterrupted flow of program execution the special 16-bit program counter (PC) register is simply incremented to the next instruction in memory. When a jump or branch is made to a different address, that address is loaded into the program counter and the CPU then carries out the instructions in sequence from that address.

Subroutines are called by a 'CALL nnnn' instruction where 'nnnn' specifies the address of the subroutine. Remembering that a GOSUB command in Basic returns to the next command after the GOSUB when a RETURN command is executed, similarly, a return to the next instruction is made from a M/C CALL instruction when a RET instruction is encountered.

RET instructions in M/C differ from the Basic RETURN command by having a number of conditional variations - see details about the M/C RET instruction in Chapter 13.

When a 'CALL' is made to a subroutine in M/C, the address of the next instruction after the call instruction is 'pushed' onto the stack. The stack is last-in/first-out temporary storage in RAM, the stack pointer (SP) register pointing to the current top of stack address. See discussion of 16-bit registers in Chapter 5.

When the CPU encounters a RET instruction at the end of a subroutine, it retrieves the return address by 'popping' it off the stack and loading it into the program counter. Program execution then continues from the next instruction after the CALL instruction as desired.

10.02 CONDITIONAL TESTS

In addition to the unconditional jump - 'JP address', the Z80 provides conditional branching.

When executing a conditional branch the CPU tests the condition of one of the bits in the flag register and, depending on the result, either branches to another part of the program or continues on to the next instruction after the conditional branch instruction.

The bits that can be tested are the zero, carry, sign and parity/overflow bits.

Table 10.1 contains the conditional branch instructions for testing each bit which causes a jump if the conditions are met.

Table 10.1

INSTRUCTION		CONDITION (if there is...)
JP	Z,address	a zero result (Z=1)
JP	NZ,address	a non-zero result (Z=0)
JP	C,address	a carry (C=1)
JP	NC,address	no carry (C=0)
JP	M,address	a negative result (S=1)
JP	P,address	a positive result (S=0)
JP	PO,address	odd parity result (P/V=0)
JP	PE,address	even parity result (P/V=1)

The above instructions contain the absolute address for the jump, but in addition the Z80 has relative addressing instructions.

10.03 RELATIVE BRANCHING

With a relative branching instruction a single offset displacement byte is specified as part of the instruction instead of a two-byte absolute address. As mentioned before relative addressing is useful for creating programs which can be loaded to different parts of memory without modification because they do not contain any absolute address references.

There is a limitation, however, because only the zero and carry flag bits can be tested with the conditional relative branch instructions.

In addition to the conditional relative branching instructions there is an unconditional relative branch instruction :- 'JR displacement'.

Table 10.2 contains the conditional relative branchng instructions.

Table 10.2

INSTRUCTION		CONDITION (if there is...)
JR	Z,displacement	a zero result (Z=1)
JR	NZ,displacement	a non-zero result (Z=0)
JR	C,displacement	a carry (C=1)
JR	NC,displacement	no carry (C=0)

10.04 CALCULATING THE DISPLACEMENT BYTE

When the CPU encounters a relative branch instruction and needs to execute the relative jump, the displacement value is added to the current program counter value to work out the address for the jump.

The displacement value is worked out by counting the number of bytes between the starting address of the next instruction after the relative branch instruction and the start of the instruction to be branched to.

The start of the next instruction after the relative branch instruction is taken as 0 because the program counter is sitting at this address after it has fetched all of the branch instruction bytes.

As an example we will look at the calculation of the displacement byte in the short graphics program in the previous chapter. Refer to Listing 10.1 for the assembly source.

Listing 10.1

LINE	LABEL	MNEMONIC		ADDRESS	CODE
0001	STRT	LD	HL,7000H	8000	21 00 70
0002		LD	BC,0200H	8003	01 00 02
0003	LOOP	LD	A,(7921H)	8006	3A 21 79
0004		LD	(HL),A	8009	77
0005		INC	HL	800A	23
0006		DEC	BC	800B	0B
0007		LD	A,B	800C	78
0008		OR	C	800D	B1
0009		JR	NZ,LOOP	800E	20 F6
0010		RET		8010	C9

Starting from address 8010H - the start of the next instruction after the relative branch instruction in line 009 - count the number of bytes back to, and, including the start of the instruction in line 003. Remember to count the byte at 8010H as 0. The number counted is 10 bytes. Because the movement is backwards then the displacement is -10.

There is no direct representation of negative numbers in binary, so a possible solution could be to use the first seven bits (bit 0 to bit 6) to represent the quantity and the eighth bit (bit 7) to indicate whether the quantity is positive or negative.

10.05 TWO'S COMPLEMENT NUMBERS

Unfortunately, in order to simplify the logic in the design of Arithmetic Logic Units inside CPUs, a different system called "two's complement" has been adopted for the representation of negative numbers. We have already dealt briefly with two's complement numbers in Chapter 8 (under 'Signed Integers').

Although the use of this system might simplify things for the chip designer, it makes things a little tricky for us.

Normally to work out the value of a binary byte we use all eight bits, but in two's complement we use the eighth bit to indicate the sign of the quantity. The eighth bit set (1) indicates a negative quantity while a 0 indicates a positive quantity, in agreeance with the convention for the sign bit of the flag register.

The remaining seven bits for a positive quantity is evaluated as for a normal binary quantity and has a maximum value of +127.

For negative quantities the eight bit is a 1, but the remaining seven bits are not simply the binary magnitude. The following is the procedure for working out the two's complement of our displacement, -10.

1. The displacement is negative so the eighth bit is a 1.

2. Invert all the remaining seven bits which represent the magnitude:-

   10 (decimal) = 0001010 (7-bit binary)

   invert lower seven bits = 1110101

3. Add one (0000001):

```
  1110101
+ 0000001
  1110110
```

4. Make up the full eight bit binary byte - 11110110

5. Break the byte into two 4-bit numbers and evaluate:-

```
1111/0110

1111 = 15 decimal          0110 = 6 decimal
     = 0FH                      = 6H
```

i.e., the two's complement of the displacement -10 is 0F6H.

The biggest negative displacement is given by the eighth bit set to a 1, and the remaining seven bits all 0s, i.e., 10000000 - which evaluates to -128.

So the maximum range for a relative branch is +127 (forwards) and -128 (backwards) and is a major restriction of the use of such instructions.

Once again the advantage of using an Editor Assembler can be appreciated as it will automatically calculate the displacement byte from the source listing.

10.06 SAVING DATA ON THE STACK

The CPU automatically uses the stack (controlled by the stack pointer (SP) register) when a call is made to a subroutine. The CPU uses the stack to store the return address which will be retrieved when the call has finished.

The stack can also be used by the programmer in a more direct way by PUSHing and POPing the contents of the register pairs, i.e., data is stored on the stack as 16-bit quantities.

The SP register is decremented or incremented by two during each stack operation because each memory location can only store eight bits, so two bytes are needed to store the 16-bit values.

Because there are a limited number of registers available, sometimes the need arises to save the contents of a register pair temporarily while the register pair is used for another purpose. The stack can be used for this purpose.

The following source code in Listing 10.2 is for the previous short graphics machine code program, modified to use the stack as temporary storage for the contents of the A register.

Listing 10.2

```
LINE    LABEL       MNEMONIC

001     STRT        LD          HL,7000H
002                 LD          BC,0200H
003                 LD          A,(7921H)
004                 PUSH        AF
005     LOOP        POP         AF
006                 PUSH        AF
007                 LD          (HL),A
008                 INC         HL
009                 DEC         BC
010                 LD          A,B
011                 OR          C
012                 JR          NZ,LOOP
013                 RET
```

Three lines have been added, two PUSHes to the stack and one POP. The PUSH to the stack in line 004 is quite straight-forward, the contents of the A register just loaded from location 7921H is being saved to the stack because it is going to be overwritten in lines 010 and 011. However, the 'POP AF - PUSH AF' sequence in lines 005 and 006 might look a little odd.

Well, in fact, the first time the program executes these instructions they are really not necessary because the contents of the A register are not destroyed until further down. The reason for writing the program this way is to ensure the correct sequence of operations for the repetitive loop operations between lines 005 to 012 later on.

Coming into the loop at line 005, the required contents of the A register have been saved to the stack. The first time lines 005 and 006 are executed they are not really necessary, but when the program loops back from line 012 the contents of the A register have been destroyed in lines 010 and 011. Line 005 renews the contents of the A register by POPping the saved value off the stack.

Remember, the execution of a POP instruction copies the contents of the stack pointed to by the SP register. Then the SP register is incremented to the next last saved value (the stack grows **down** in memory when items are added). This means that another POP will not return the same value because it will be retrieved from the next position on the stack.

So we actually have to resave the value to the stack ready to be POPped the next time around. This is the purpose of line 006.

By the way, when a value is POPped off the stack, the entry in the stack memory is not destroyed, the SP register is just incremented to the next entry. Therefore, in theory, you could just as easily reposition the SP register back to the value by decrementing the SP twice. I say in theory, because such a practice is very dangerous as it is very easy to lose track of what is what by using this technique. I have not seen this used, nor have I ever used it.

Some thought about what happens to data during stack operations is useful.

When a register pair is PUSHed, the contents of the register pair are copied to the next two lower locations below the last entry to the stack. The contents of the register are not destroyed, but simply copied to the stack.

When a register pair is POPped, the current data value on the stack overwrites the contents of the register pair. The previous value in the register pair is destroyed. The SP is incremented two bytes higher up in memory to the next value held on the stack.

Sometimes more than one register is needed to be saved on the stack. For example, if you are calling a ROM routine which uses several registers, you might need to save the contents of those registers until the execution returns from the call. The source code in Listing 10.3 below shows such a situation.

Note how the registers are POPped in strict reverse order to the order in which they were PUSHed. This is to ensure that the correct values go into the right registers, as the stack is a LIFO stack. The term LIFO comes from Last In - First Out.

Listing 10.3

```
LINE    LABEL     MNEMONIC

001     STRT      PUSH    AF
002               PUSH    BC
003               PUSH    DE
004               PUSH    HL
005               CALL    ROM
006               POP     HL
007               POP     DE
008               POP     BC
009     END       POP     AF
```

We can use a variation in the order of PUSH-ing and POP-ing as a convenient way of doing a 'switch-a-roo' on the contents of register pairs. The sequence below swaps the contents of the BC and DE register pairs.

```
PUSH    BC
PUSH    DE
POP     BC
POP     DE
```

Juggling of register data on the stack is one of the more tricky, but nonetheless useful, techniques of machine code programming.

The next chapter on the VZ Editor Assembler shows how it is used to simplify the entry of machine code programs by using mnemonics in assembly language source code.

-ooOoo-

CHAPTER 11

EDITOR ASSEMBLER

The VZ Editor Assembler is a simple means of editing, assembling and creating auto-execute M/C programs for the VZ. The Editor Assembler is not the most sophisticated that I have used, but is perfectly adequate for the purpose for which it was intended, that is, to provide a simple means of assembling and creating M/C object tapes for the VZ.

The only serious limitation I have found with it is the small size of memory available for editing and assembling the programs. This is because the Assembly Source code (mnemonics) and the Object Code (actual machine executeable code) are co-resident in a memory area of approximately 11 Kbytes - about 21 Kbytes in later versions.

11.Ø1 DEFINITIONS

The VZ Editor Assembler is what is called a 'two pass' assembler. The term 'two pass' refers to the fact that the Assembler scans the Assembly Source code twice. Each scan is called a pass.

During the first pass, labels are read and relative offsets are calculated which are used during the second pass. The second pass decodes opcodes, operands and expressions.

11.Ø2 ASSEMBLY LANGUAGE SYNTAX

An assembly language program, or a source code listing, consists of labels, opcodes, pseudo-ops, operands, and comments in a sequence which defines the user's program. These labels, opcodes, pseudo-ops, and operands must be separated from each other by one or more ASCII commas or spaces.

Comments are separated from the rest by a separate line beginning with a semi-colon. The following illustrates the source code format:

```
================================================================
(labels)       opcode        [operand 1 {,operand 2}]
================================================================
```

11.03 LABELS

A label is composed of one to four characters. The characters used in a label cannot be any of the non-printable ASCII characters, an ASCII blank, or any of the following characters:-

! " # $ % & ' ( ) @ = - / ? + * ; : < > . ,

As well, the first character of a label cannot be a decimal digit. All labels must begin in column one. A label can be used in any line in the source code, but cannot appear twice in the label field.

A label in the label field of a source code listing is set to the current 16 bit value of the assembler program counter. The exception to this is when the label is assigned a 16 bit
! " # $ % & ' ( ) @ = characters:-
value by the pseudo-op 'EQU'. Once defined, a label may be used in other fields in source lines elsewhere in the source code listing.

11.04 OPCODES

In the Z80 instruction set there are 67 opcodes, such as 'LD'; 25 operand keywords, such as 'HL'; and nearly 700 legal combinations of opcodes and operands recognised by this assembler. In this manual they are only briefly documented, so it is essential that a companion Z80 'how to program' text is obtained. The are also many undocumented combinations of opcodes and operands, but they are for the advanced programmer, and so do not concern us here.

11.05 PSEUDO-OPS

The VZ Editor Assembler recognises four pseudo-ops. These appear in the opcode field of a source statement. Labels for these pseudo-ops are optional except for the 'EQU' pseudo-op. The pseudo-ops do not necessarily generate object code, as all opcodes do, but can cause certain values to be loaded into certain bytes, or can reserve bytes. All the pseudo-ops direct the Assembler to cause some action to happen.

A pseudo-op can be used to assign a value to a label. This is the 'EQU' pseudo-op and has the format:-

```
label    EQU      nn
```

where nn is a 16 bit value.

For example:-

```
VIDE     EQU      7000H
```

here the label 'VIDE' has been EQUated to 7000H, the beginning of video RAM memory, and can be used later on in the source code to save referring to the hexadecimal address 7000H directly. For example:-

```
STRT    LD      HL,VIDE
```

could then be used in place of:-

```
STRT    LD      HL,7000H
```

This value is fixed by the 'EQU' pseudo-op and cannot be redefined by another pseudo-op or appear in the label field of another statement in the source code.

The label can also be made equal to the address of the label itself which is the current value of the assembler program counter. The format is:-

```
 label  EQU     $
```

where $ is the current value of the assembler program counter.

Particular bytes within a program can have their contents defined by one of two pseudo-ops, as opposed to having their contents determined by the assembling of a source line and the resultant object code.

The first of these is the 'DEFB' pseudo-op which has the following format with the brackets around the 'label' indicating that a label is optional.

```
(label) DEFB    n
```

where n is an 8 bit value which becomes the contents of the byte in memory at that address. ASCII characters can be assigned to these bytes by enclosing them in quotes. For example:-

```
(label) DEFB    "A"
```

The other pseudo-op is the 'DEFW' pseudo-op.

```
(label) DEFW    nn
```

where nn is a 16 bit value. The least significant byte of the 16 bit quantity nn is loaded into the address of the current program counter, and the most significant byte of nn is loaded into the address of the program counter + 1. These two consecutive bytes form what is called a 'word'.

If the user wants to reserve a certain section of RAM inside the program area but does not want to initialise it with specific values during assembly, the 'DEFS' pseudo-op can be used. The format is:

(label) DEFS nn

where nn is a 16 bit value. Using this pseudo-op reserves nn bytes of memory and caused the assembler program counter to be incremented over this area.

11.06 OPERANDS

There may be zero, one, or two operands present in the source statement depending on the opcode or the pseudo-op used. An operand can take one of the following forms: - a generic operand, a constant, a label, the program counter symbol '$' or an expression.

A generic operand is a keyword which has special meaning to the Assembler. These keywords are recognised as having only one meaning and should not be used as labels. Below is a list of these operands and their meanings.

OPERAND	MEANING
A	A register (accumulator)
B	B register
C	C register
D	D register
E	E register
F	F register (flags)
AF	AF register pair
AF'	AF' register pair
BC	BC register pair
DE	DE register pair
HL	HL register pair
SP	stack pointer register
$	program counter
I	I register (interrupt vector MS byte)
R	refresh register
IX	IX index register
IY	IY index register

OPERAND	MEANING
NZ	non-zero
Z	zero
NC	non-carry
C	carry
PO	parity odd/non overflow
PE	parity even/overflow
P	sign positive
M	sign negative

A constant used as an operand must be in the range 0 through 0FFFFH or 0 through 65535. The are two types of constants, but the default is decimal. Hexadecimal numbers must start with a digit 0 to 9 and end with 'H',e.g., 22H, 0FCH, 15H, 0A3H.

ASCII constants are characters enclosed in double quotes and are converted to their equivalent ASCII code byte ("A"= 41H).

Labels may be used as operands with two conditions: firstly they have to appear in the label field of a source line elsewhere in the source listing, and secondly, labels cannot be defined by labels which are not defined themselves previously. This is an inherent limitation of a two pass assembler.

The symbol '$' can be used as an operand. It represents the current value of the assembler program counter (the address at which the code is currently being assembled to).

Finally, the allowed range for the offset in a 'JR' instruction is -128 to +127.

11.07 TYPICAL EDITING/ASSEMBLING SESSION

To illustrate the actions in a typical Editor Assembler session, we will use the example of the simple half screen clearing routine from a previous exercise. The process can be broken down into a number of steps:-

1. Write down on paper a description of the process that is required to be implemented in your program.

2. Draw a flowchart of each sub-task in the process, and if the program is sufficiently complex, draw an overall general flowchart to show how the various sub-task modules link together.

3. Write down on paper the assembly language mnemonics which implement the operations of the flowchart. I recommend you don't type the source code directly into the Editor Assembler without writing it down on paper first. This is because in spite of the convenience of being able to quickly edit the source on the screen, the video screen has not been able to replace the facility of seeing the whole program before you at one time (spread out across the table in front of you). This 'panoramic' paper view wins hands down when it comes to making the program a nicely integrated unit. electronic 'cut and paste' will still come a poor second to this approach until we have video displays which occupy the size of a table top.

4. Once you are satisfied with the logic and the flow of the code, type it into the Editor Assembler, using the 'I' command.

For example, enter the small routine in Listing 11.1 from a previous example:-

Listing 11.1

```
001    ; HALF-SCREEN CLEAR ROUTINE
002    START LD    B,0FFH
003          LD    HL,7000H
004          LD    A,60H
005    LOOP  LD    (HL),A
006          INC   HL
007          DJNZ  LOOP
008          RET
```

Always make the first line of your source code a comment line (must be started with a ';'), perhaps a short one line description of the function of the program.

If you put actual program mnemonics on the first line you will find that you will not be able to insert extra code easily at the head of the code later. This is because if you enter 'I1' when you want to insert a line at the head of the source code you will find the Editor Assembler will respond by bringing up the second line.

If you type 'I0' you will be presented with the next after the last line of the source code. Therefore, fill the first line with something that you will not be likely to alter often, e.g. a title for the program.

You can of course insert lines at line 001 by typing 'I1' (the Editor Assembler responds by bringing up line 002) and then re-typing line 001 into line 002. You can then enter the new code in line 001 by entering 'E1'.

If it is a long listing I suggest you save it periodically on tape using the 'TS' command. When it comes to deciding how often you 'back-up' or save your source code, decide on the maximum amount of time you want to waste if the power should accidentally go off. If you don't mind losing half an hours' work, then only back-up every half hour, but remember - it is not just the typing time you have lost. Sometimes a considerable amount of working out and design thought is lost and you might have to go back and spend time trying to remember why you did something a certain way.

Once again, if you write down the program source code on paper before you begin to type it into the Editor Assembler, you are ahead (or at least, not too far behind).

5. Set status to printer output using 'S1C' and get a printout of the source code for use during debugging.

6. Set the start address for assembly by using the 'O' command, and use the 'A' command to assemble the source code after setting the status to 'S1A'. Assuming you have set the origin to 8000H (32768 decimal) either by 'O8000H' or 'O32768', the Editor Assembler will produce the following:-

Listing 11.2

```
001 ; HALF-SCREEN CLEAR ROUTINE
002 STRT LD    B,0FFH
8000 06 FF
003      LD    HL,7000H
8002 21 00 70
004      LD    A,60H
8005 3E 60
005 LOOP LD    (HL),A
8007 77
006      INC   HL
8008 23
007      DJNZ  LOOP
8009 10 FC
008      RET
800B C9
```

7. If there are no errors in the assembly go to step 9, otherwise set the status to 'S1C' to get a printout of the errors.

8. Go back to your source code listing with this error listing, find the errors, correct them, and then produce a new source listing. Write a cancellation note across the old source to avoid the possibility of confusion over which is the the latest, most correct version of the listing. Now go back to step 6 and repeat the process until all errors have been eliminated. As long as you are keeping track of the latest correct source listing there is no need to update the source on tape, unless, of course, the corrections are very drastic, or the listing is very long.

9. At this stage you should have an error-free source code listing, however, you may still have a long way to go. This is because the errors found by the Editor Assembler are simply syntax-like errors.

There are now basically three paths to follow to fully check out your program code.

The first is to assemble the code into RAM just above where the source code is held using the 'R' command. This has the advantage of being a quick way of checking out the code, but has the disadvantage of not being able to assemble the code at the position in memory where you want the program to run. I have rarely used this facility. Also, if the program has a bug, it can overwrite the Editor Assembler object code, or lock up the computer completely, requiring you to switch off the VZ to regain control. In either case you will have to reload both the Editor Assembler and the offending source code in order to correct it.

The second path available is to create an object tape of the program. This allows the program to be loaded where it is supposed to run finally. However, it is necessary to reset the VZ in order to load the object tape and run it. Once again you will have to load the Editor Assembler and the source code back in to correct any errors, and then create a new object tape. This will have to be repeated for every error found. In other words, it is impossible to easily make any corrections to the program without loading back in the Editor Assembler and the source code.

The third path is the one I tend to use in general, and that is to take the assembled object code and convert it to decimal notation and place it in 'DATA' statements of a Basic loader program. I find this a handy method for testing small routines for a larger program, or small routines to be called from a main Basic program, using the USR(Ø) Basic command as described previously.

The small half screen clearing subroutine was developed using this approach. If you study line 2010 (repeated here below) and examine each of the entries in that line you will find they are the decimal equivalents of the hexadecimal object code produced by the Editor Assembler when it assembled the above source code.

```
2010 DATA 6,255,33,0,112,62,96,119,35,16,252,201,-1
```

The final -1 is an 'end of data' flag for the loader.

WARNING ! WARNING ! (Don't say you haven't been warned)

As it says in the Editor Assembler Instruction Manual, be sure to save your source code before you attempt to run your program in memory to avoid the mind-boggling, dog-kicking frustration of losing all your hard work if the program crashes. In fact, I always save my source before assembling it, that is, during the time I am entering the source code.

Appendix 8 contains the list of mnemonics and pseudo-ops recognised by the VZ Editor Assembler.

-ooOoo-

NOTES

CHAPTER 12

PROGRAMMING TECHNIQUES

This chapter will deal with the process of writing simple machine code programs. The general approach described here is applicable to either using a Basic loader or an Editor Assembler.

Writing machine code programs involves implementing the task to be performed by using the available instructions and assembling them into a program.

Compared to Basic, machine code requires a large number of instructions even for simple tasks, so it is important to find ways of reducing the number of necessary instructions in order to minimize the size and complexity of the program.

12.01 SUBROUTINES

As a Basic programmer, you should be familiar with the concept of subroutines. Just as you can write a separate section of Basic code and then 'call' it by the use of the Basic GOSUB command, in machine code program there is a 'CALL' instruction. Likewise, when a return to the original section of code is required, there is a 'RET' instruction (like the RETURN instruction in Basic).

Subroutines save the duplication of sections of code which are used a number of times in the program, thereby saving on the number of bytes needed in memory to store the program. Also, by breaking down the operation of the program into smaller, indentifiable sections, the program is easier to write, understand and correct.

Unfortunately, there is one major difference between Basic and machine code subroutines; when a call is made to a subroutine in Basic it is to a certain line number, while in machine code the call is made to a absolute memory reference.

That is, you need to know exactly where in memory the subroutine is stored. The actual physical memory address must be included in the operation code for the 'CALL' instruction (which is 0CDH followed by the low order byte, then the high order byte, of the memory address of the start of the subroutine). In Basic the line number is just a reference to a position in the Basic code listing - it has nothing directly to do with the actual physical memory address.

That means, for machine code, we have to keep track of any changes, insertions or deletions which may affect the actual memory address of our routines.

If we are assembling our machine code by hand, this becomes a tedious and error-prone operation. This is another reason for using an Editor Assembler, because this utility allows us to specify the start address of the routine by a label and then the actual address for the CALL instruction is automatically calculated by the Editor Assembler when it assembles the source code for the program.

12.Ø2 LOOPS

In Basic, the use of FOR...NEXT or IF...THEN GOTO commands allows us to construct loops and repeat sections of program, using changing variables, until certain testable conditions occur. In machine code, the use of the conditional JUMP instruction (with testing operations) can be used to construct a loop.

If we use JR instructions, these loops can be implemented using code that is relocateable. This limits the variety of tests that can be made as well as the number of bytes contained within the loop, but this is usually not a problem.

12.Ø3 FLOWCHARTS

The use of flowcharts is the sign of a competent programmer who takes a professional approach to programming. It's a case of 'real programmers DO use flowcharts'. Some programmers are of the opinion that because of the sophisticated editing utilities available, all program development, writing and checking can be done at the keyboard. Although this may be possible for simple programs, it can be shown that over a period of time such an approach is inefficient and unprofessional.

It seems to me that those who advocate this approach have never written, debugged or modified a sizeable program (> 1ØK bytes), or at least done it efficiently. Their programs can at times resemble machinery held together with fencing wire; they work, but any attempt to modify or adapt them results in a mess.

The first step in writing machine code is to set out in flowchart form the operational flow of the proposed program. Even if you have managed to write programs in Basic without flowcharts, do not make the mistake of thinking you can get away with it when writing machine code except for the simplest of programs (5Ø bytes or so).

The idea behind flowcharts is to give you an overall picture of the operation of the program in order to allow efficient writing, altering and debugging. You simply can't keep this overall picture in your head (except for simple programs).

If the program is large (and therefore complicated) begin by writing down in English a description of the performance that is required from the program. Then isolate and write down the separate sub-tasks that need to be performed.

Devising flowcharts for these sub-tasks will allow you to decide what registers to use, where data will be stored, and what routines can be used more than once in the program flow if they are made to be general in nature.

The use of flowcharts can also help you to decide the most efficient use of the available instructions, sometimes considerably shortening the code necessary.

If you can have the patience to lay out your program properly in flowchart form, you will find it a good long-term investment for future programming activity. If you include in the flowchart such details as the use of registers, what is on the stack at each point, you will find the program not only easier to debug, but also many of the clever little tricks you have devised can be re-used at a later date in other programs.

12.Ø4 EXAMPLE PROGRAM

We will write a small routine which is commonly required in many programs, that of clearing the video screen. (Although in this example only half of the screen will be cleared).

The process will be done in three stages:-

1. Identifying parameters that will be used by the routine, i.e. the start and finish of the video screen in memory to be cleared , and the character used to fill this area to produce a blank screen.

2. Drawing a flowchart to describe the operation of the routine.

3. The implementing of the tasks as described in the flowchart in assembly language and the production of machine code bytes (object code) by assembling the assembly language source code.

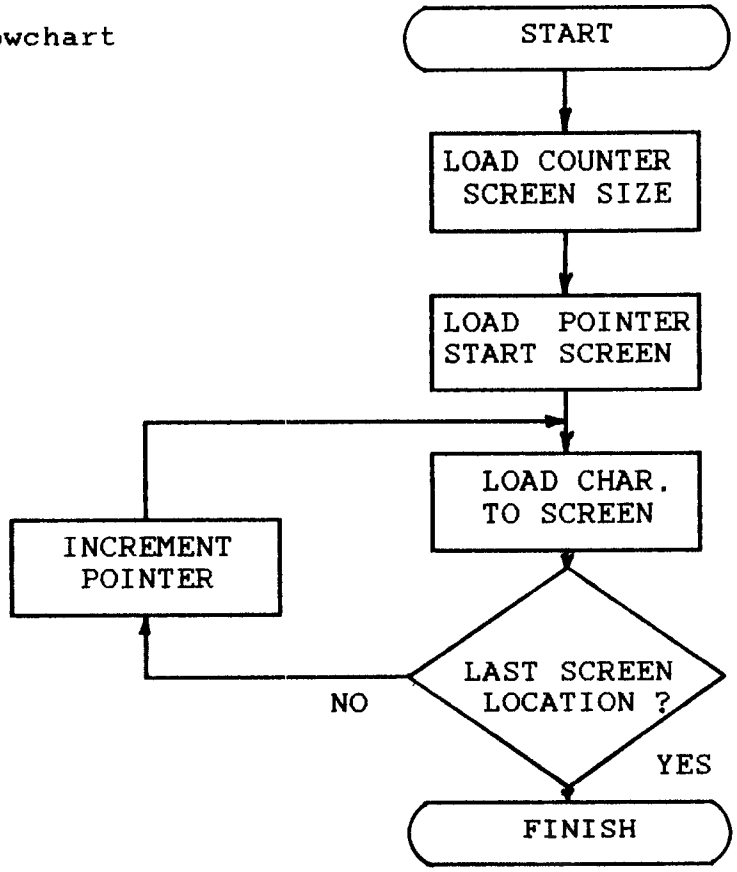
In addition, a Basic program listing is given to show the equivalent operations in Basic.

12.05 PROGRAM PARAMETERS

The start of the video text screen (MODE 0) is at 7000H (28672) and extends upwards in memory for 512 bytes to 71FFH (29183). The character code for a blank or space is 40H for a black space, or 60H for a white space.

Figure 12.1

Flowchart



Listing 12.1 Assembly Language Source Code Listing

```
START LD    B,0FFH      ;load B Register 1/2 screen
      LD    HL,7000H    ;load HL Register with start
      LD    A,60H       ;load A register with space code
LOOP  LD    (HL),A      ;load space to screen
      INC   HL          ;increment to next location
      DJNZ  LOOP        ;decrement B Register and check to see
                        ;if it is zero
                        ;if not zero go back to 'LOOP'
      RET               ;return to calling code.
```

If you use the VZ Editor Assembler to produce the object code, leave out the comments, and remember that labels have a maximum of four characters.

Listing 12.2 VZ Assembler Source and Object Code

LINE	ASSEMBLY SOURCE CODE			ADDRESS	OBJECT CODE
001	STRT	LD	B,0FFH	0000	06 FF
002		LD	HL,7000H	0002	21 00 70
003		LD	A,60H	0005	3E 60
004	LOOP	LD	(HL),A	0007	77
005		INC	HL	0008	23
006		DJNZ	LOOP	0009	10 FC
007		RET		000B	C9

12.06 ASSEMBLY LANGUAGE SOURCE CODE LISTING

The above listing is what will be produced by the VZ Editor Assembler with the exception that the Address and Object Code will be on the next line after the Source Code. I have patched my Editor Assembler to output the Address and Object Code on the same line (only for hard copy to the printer).

The equivalent Basic program would be:-

Listing 12.3

```
100 B=255
200 HL=28672
300 A=96
400 POKE HL,A
500 HL=HL+1
600 B=B-1:IF B<>0 THEN GOTO 400
700 END
```

Note how the hexadecimal quantities in the Assembly Language Source code have been converted to their decimal equivalents and the variables used in the Basic program are the same as the register names in the Assembly Source code for ease of comparison.

Finally, to clearly see the difference in speed between Basic program execution, use the Basic M/C loader to load and execute the machine code program above and compare the time to clear the top half of the screen. The two (Basic and M/C) clearing routines have been combined into the one program.

Listing 12.4 is the Basic screen clearing program (lines 30-800) to which a M/C loader has been added (900-2010).
This example illustrates two things clearly. The first is the vast superiority of M/C over Basic as regards to speed, and how small M/C subroutines can be conveniently called from a main Basic program to perform those functions which require speed, while the main Basic program is used to output text and accept input from the user via the keyboard.

Once again, I put forward the idea that there is no need to write all your program in M/C in a computer which has a resident Basic Interpreter.

Listing 12.4 The Screen Clearing Program

```
30 REM BASIC VERSION
40 CLS
50 FOR I=1 TO 16:PRINT"****************";:NEXT I
60 INPUT"HIT 'RETURN' TO START";D$
70 PRINT@416,"STARTING BASIC CLEAR SCREEN"
100 B=255
200 HL=28672
300 A=96
400 POKE HL,A
500 HL=HL+1
600 B=B-1:IF B<>0 THEN GOTO 400
700 PRINT@448,"FINISHED"
800 INPUT"HIT 'RETURN' TO GO TO M/C";D$
900 REM M/C VERSION
1100 CLS:ST=32500
1200 FOR I=1 TO 16:PRINT"****************";:NEXT I
1300 READ D:IF D=-1 THEN GOTO 1600
1400 POKE ST,D
1500 ST=ST+1:GOTO 1300
1600 MB=INT(32500/256):LB=32500-MB*256
1700 POKE 30862,LB:POKE 30863,MB
1800 INPUT"HIT 'RETURN' TO START";D$
1900 PRINT@416,"STARTING M/C CLEAR SCREEN":X=USR(0)
2000 PRINT@448,"FINISHED":END
2010 DATA 6,255,33,0,112,62,96,119,35,16,252,201,-1
```

-ooOoo-

CHAPTER 13

THE Z-80 INSTRUCTION SET

As mentioned before, when you are creating your machine code program it is very useful to write it down in the form of mnemonics. These mnemonics are a kind of shorthand for the instructions that can be utilised by the CPU. There are several very good reasons for using this shorthand representation:

It enables us to:-

1. 'verbalise' the operations we wish to carry out and we can fairly easily step through the code to check that everything is correct. Imagine trying to step through a list of op-codes expressed in numerical form (e.g. 3EH, 22H, 23H, 0AFH, 0EDH etc.) in order to check it out (debug it).

2. express some quantities in general terms while we are still in the stages of initially writing our code so that when we change the program we do not have to recalculate those quantities.

3. Finally, if we are using an Editor Assembler (you should if you are serious about your programming) we do not need to actually write down the numerical values of each byte of the machine code program - the Editor Assembler does it for us. Also, if we insert a new piece of code between existing code, the Editor Assembler automatically adjusts for the new locations when it re-assembles.

13.01 MNEMONICS

The mnemonics used to represent the various instructions are designed to remind us of the operation that the instruction carries out and so a generalised form of description is used. The following illustrates the general source code format:-

(label) op-code [operand 1{,operand 2}]

As indicated by brackets, some of the parts of the source code line are optional.

The 'label' field is used to mark a particular point in the source code so that a jump can be made to this point from some other part of the code. In other words it has the same function as the line number has in a Basic 'GOTO line number' statement.

The 'op-code' field must be included. The letters used here define the basic function of the instruction, e.g. LD means load, INC means increment, ADD means add (surprise, surprise!!), while ADC means add with carry and so on.

Letters which represent the registers (AF, BC, HL etc.) do not appear in the op-code field itself (except for several direct Accumulator operations).

The 'operand 1,operand 2' field generally indicates what item the operation will be carried out on. The operands might specify a register, a memory location or an actual numerical value. Note that 'operand 2' is an extension of 'operand 1', i.e. you can't have 'operand 2' by itself.

In any list of instructions for a CPU, there are some instructions which are almost identical to some others, except for referring to, say, a different register. In order to avoid repeating these almost identical instructions in the following descriptions, we can replace the specific references by general terms.

In the section below the following general conventions are used:-

r — Any individual 8-bit register (A, B, C, D, E, H and L.)

rr — 16-bit register pair. Will always mean BC, DE and HL. The description will say if the other 16-bit registers are also included (AF, IX, IY and SP).

n — 8-bit data byte.

nn — 16-bit data byte or memory address

d — 8-bit offset or relative displacement. Has the range of +127 to -128.

When brackets are used around a 16-bit quantity, that quantity is not the actual value to be used, but rather is used as a 'pointer' to a memory location where the data to be acted upon is stored.

Even more obscure is the situation when brackets surround a register pair designation. Here the register pair contains the 16-bit quantity which is used as the pointer to the memory location which contains the data to be acted upon.

For example:-

```
LD        A,(7000H)
```

means load the Accumulator (8-bit register A) with the 8-bit data stored in location 7000H, while:-

```
LD        A,(HL)
```

means load the Accumulator with the 8-bit data stored in the memory location pointed to by the 16-bit address held in the HL register.

This seemingly indirect way of specifying the location of a byte in memory is, in fact, very useful. For example, when we want to scan through a block of memory and test for a particular value.

Instead of having separate instructions to reference each memory location to be scanned, we can load the first memory location into the HL register, use 'LD A,(HL)' to load the data byte into the Accumulator and test it. To access the next byte in memory, we simply INCrement the HL register and repeat 'LD A,(HL)' etc., using a loop construct (much the same way as a FOR...NEXT loop in Basic).

13.02 ACCUMULATOR OPERATIONS

The Accumulator, or A register, is the only 8-bit register in which 8-bit arithmetic operations can be performed. The result of the operation is written back into the A register, overwriting values previously stored in the Accumulator.

```
ADD A,n          :the next 8-bit data byte 'n' following the
                  instruction byte in program memory is added
                  to the contents of A.

ADD A,r          :the contents of register 'r' are added to A.

ADD A,(HL)       :the contents of the memory location pointed
                  to by HL are added to A.

ADD A,(IX+d)     :the contents of the memory location pointed
ADD A,(IY+d)      to by adding the displacement 'd' to the
                  contents of IX and IY respectively, are added
                  to A.
```

The previous instructions are repeated for ADC (add with carry), except the condition of the carry flag is included in the sum, i.e.:-

ADC A,n : ADC A,r : ADC A,(HL) : ADC A,(IX+d) : ADC A,(IY+d)

Similarly, for 8-bit subtraction:-

SUB n : SUB r : SUB (HL) : SUB (IX+d) : SUB (IY+d)

and with carry:-

SBC n : SBC r : SBC (HL) : SBC (IX+d) : SBC (IY+d)

Note that for the SUB and SBC instructions the letter 'A' is not included as operand 1 as is the case for ADD and ADC.

Three of the above 8-bit arithmetic operations are available for 16-bit operations, with the HL register acting as the 16-bit 'accumulator', i.e. the result is loaded back into HL.

ADD HL,rr :the contents of register pair 'rr' are added to HL, where 'rr' may be BC, DE, or HL itself (where it doubles the value in HL) or SP.

ADC HL,rr :the contents of register pair 'rr' (as above) and the carry bit of the flag ('F') register are added to HL.

There is only one 16-bit subtraction operation available; subtract with carry:-

SBC HL,rr :the contents of the register pair 'rr' and the carry bit are subtracted from the HL register.

Note that if you want to do a 'subtract without carry' operation then precede the 'SBC HL,rr' instruction with an 'XOR A' which has the effect of zero-ing the A register and clearing the carry flag.

Two more ADDs are available which operate on the index registers IX and IY. They are:-

ADD IX,rr :where 'rr' may be BC, DE, IX itself, or SP.
ADD IY,rr :where 'rr' may be BC, DE, IY itself, or SP.

Another group of instructions which use the Accumulator are the LOGICAL or BOOLEAN operations. The operations are carried out on a bit-by-bit comparison basis between the eight bits in the accumulator and the identical eight bits of another specified byte.

The result of each bit comparison is loaded back into the corresponding bit position in the Accumulator.

Firstly, let's look at the logical AND operation. Here the result for each bit position is logic 1 if the bit in the Accumulator is 1 AND the bit in the comparison byte is 1. Otherwise the result is Ø.

AND n	:the Accumulator byte is ANDed with 'n' and the result loaded back into the Accumulator.
AND r	:the contents of the Accumulator and the 'r' register are ANDed and the result is left in the Accumulator.
AND (HL)	:the contents of the memory location pointed to by the HL register and the Accumulator are ANDed. Result in A.
AND (IX+d) AND (IY+d)	:the contents of the memory location pointed to by adding the displacement 'd' to the contents of IX and IY respectively, are ANDed with the contents of the A register.

The logical OR operation is similar to the AND by being a bit-wise comparison operation. Here the bit result is 1 if either, or both of the A register and comparison byte bits are 1. The OR instructions are of the same form as the AND instructions, i.e.:-

OR n : OR r : OR (HL) : OR (IX+d) : OR (IY+d)

The logical XOR (eXclusive OR) operation is once again similar to the other two logical operations. For the XOR, the result is 1 if one (and ONLY one) of the bits in the A register or the comparison byte is 1, i.e., each bit result is set to 1 if the two bits compared are different.

Once again the form of the XOR instructions is similar to the ADD and OR instructions:-

XOR n : XOR r : XOR (HL) : XOR (IX+d) : XOR (IY+d)

Incidently, for all the above logical operations the carry flag is set.

Two instructions to round off the logical operations on the Accumulator.

CPL :All bits are logically inverted, i.e. if a bit was 1 it is made 0, and vice-versa.

NEG :the contents of the A register are subtracted from zero.

13.03 LOAD INSTRUCTIONS

The largest and most frequently used group of instructions are LOAD instructions. This is not surprising because loading instructions are used to initialise registers or memory locations, and move data around inside the computer (from register to register, register to memory and back, etc.). In general, the item given in operand 1 is the DESTINATION item, while the item given in operand 2 is the SOURCE item.

Firstly, the 8-bit register loads:

LD r,r :load first 'r' with the contents of the second 'r'.

LD r,n :load 'r' with the eight-bit value 'n'.

LD r,(HL) :load 'r' with the contents of the memory location pointed to by HL.

LD r,(IX+d)
LD r,(IY+d) :load 'r' with the contents of the memory location pointed to by adding 'd' to the contents of IX and IY respectively.

In addition to the general register loads given above, the A register can be loaded from an address actually contained in the instruction itself (called immediate addressing). Also the memory location from which the data is to be loaded into the A register can be pointed to by the contents of the BC and DE registers.

LD A,(nn) :load A with the contents of memory location 'nn'.

LD A,(rr) :load A with the contents of the memory location pointed to by the contents of the register pair 'rr'.

To round off the 8-bit register loads, note that every load instruction which fetches a value from a memory address has its storage equivalent (EXCEPT for data loads from bytes within the instruction itself).

That is, we have opposites for LD A,(nn), LD r,(HL), LD A,(rr) etc., but not for LD r,n, LD A,n etc.. For example:-

LD (HL),A : LD (nn),A : LD (HL),r etc., but NOT LD n,A !.

Next, we will deal with the 16-bit register pair loads (BC, DE, HL, IX and IY).

LD rr,nn :load the register pair 'rr' with the 16-bit data 'nn'. The first byte after the instruction in program memory is loaded into the low order register of the pair (the 'L' register in the HL register pair) and the next byte in program memory is loaded into the high order register of the register pair (the 'H' register in the HL register pair).

LD rr,(nn) :load the register pair 'rr' with the contents of two consecutive memory locations, the first of which is specified by 'nn'. The byte located at memory location 'nn' is loaded into the low order register, while the byte located at location 'nn+1' is loaded into the high order register.

The one opposite instruction for 16-bit loads is:-

LD (nn),rr :load the location specified by 'nn' with the contents of 'rr' - the low order byte of the register pair being loaded into the location 'nn' and the high order loaded into location 'nn+1'.

It is also possible to load 16-bit data into memory from an instruction. The location to which the 16-bit program data is to be loaded can only be pointed to by the HL, IX and IY register pairs.

LD (HL),nn : LD (IX+d),nn : LD (IY+d),nn

Finally, we can set the SP to point to any 16-bit memory location by loading it with the contents of the HL, IX or IY register pairs.

LD SP,HL : LD SP,IX : LD SP,IY

The 16-bit contents of SP cannot be accessed directly (there is no LD HL,SP etc. instruction), however, we can access the SP indirectly by executing an ADD HL,SP instruction after we have zero-ed the HL register pair, e.g.:-

```
LD          HL,0000H
ADD         HL,SP
```

Now we have the contents of the SP register effectively loaded into the HL register pair.

13.04 JUMPS

Many of the types of constructs used in Basic programming are also needed in machine code programming. We need, for instance, an absolute jump instruction to perform the same function as 'GOTO 1000' does in Basic. We do this by loading the PC with a new address which is the address of the section of program to where we want the program execution to jump to. We cannot do this directly, but there is a more convenient and flexible way; via JUMP instructions.

First, the absolute unconditional jumps:-

```
JP nn           :load the PC with the address 'nn'. Execution
                 will continue at this new address. Remember,
                 low order byte first.

JP (HL)         :load the PC with the contents of HL, IX or IY
JP (IX)          respectively. Execution now continues at the
JP (IY)          new address pointed to by HL, IX or IY.
```

Just as we have conditional GOTOs in Basic, we can have conditional jumps in machine code. A conditional GOTO in Basic acts on the result of some test, e.g.:-

```
IF X=100 THEN GOTO 1000
```

In our Z80 machine code case the results of tests are indicated in the states of the bits in the 'F' (flag) register. They are:-

```
JP NZ,nn            :jump to address 'nn' if zero flag is 0.
JP Z,nn             :jump if zero flag is 1.
JP NC,nn            :jump if carry flag is 0.
JP C,nn             :jump if carry flag is 1.
JP PO,nn            :jump if parity flag is 0.
JP PE,nn            :jump if parity flag is 1.
JP P,nn             :jump if sign flag is positive.
JP M,nn             :jump if sign flag is negative.
```

Notice the jumps are to absolute locations (given by 'nn'). We also have RELATIVE jumps, which are so many steps back or forward from the current position. The size of the relative jump is given by an 8-bit displacement 'd'. Once again the range for this 8-bit offset is +127 to -128. The positive values of the offset cause jumps forward (or higher up) in memory, while the negative values cause jumps backwards (or lower down) in memory.

JR d :jumps 'd' number of memory bytes unconditionally.

There are only five conditional relative jump instructions:-

JR NZ,d : JR Z,n : JR NC,n : JR C,n

(NZ, Z, NC and C as above).

The fifth conditional relative jump instruction is similar to the FOR...NEXT loop construct in Basic.

DJNZ d :decrement the B register, and if it is not zero, jump 'd' number of memory bytes (usually backwards to the beginning of a loop). If B is zero continue execution at the next byte in program memory following the DJNZ instruction.

The machine code equivalent to the Basic GOSUB is the CALL instruction. When a CALL is executed, the current program memory address (the contents of PC) are pushed onto the stack, and the PC is loaded with the new address of the subroutine CALLed. We have the unconditional CALL:-

CALL nn :go to the subroutine at address 'nn'.

and conditional CALLs:-

CALL NZ,nn : CALL Z,nn : CALL NC,nn : CALL C,nn : CALL PO,nn
CALL PE,nn : CALL P,nn : CALL M,nn

To RETurn from the subroutine CALLed back to the original program memory location, we have the unconditional RETurn instruction:-

RET

and the conditional RETurns:-

RET NZ : RET Z : RET NC : RET C : RET PO : RET PE : RET P
RET M

13.05 TESTING

Testing operations affect the state of bits in the F register, which then can be used in the conditional jumps detailed above. The testing operation does not alter the values under test, but the flag bits are set/reset according to the result of the test.

A comparison operation between the contents of the A register and a specified byte can be made:-

```
CP r            :subtract the contents of 'r' from the
                 contents of A. If (A) = (r) then Z=1 (Z), C=0
                 (NC) ; if (A) < (r) then Z=0 (NZ), C=1 (C) ;
                 if (A) > (r) then Z=0 (NZ), C=0 (NC).

CP (HL)         :byte to be tested is pointed to by HL.

CP (IX+d)       :byte pointed to by IX+d, IY+d respectively.
CP (IY+d)
```

Instead of testing the whole byte in one operation, we can test individual bits of a byte with BIT instructions. For the following BIT instructions, 'bit' refers to one of the eight bits in the target byte, numbered from 0 to 7. Bit 0 being the LSB (Least Significant Bit).

```
BIT bit,r       :place the complement (inverted value) of the
                 specified bit from 'r' into the 'Z' bit of
                 the flag register.

BIT bit,(HL)    :place the complement of the specified bit
                 from the memory location pointed to by HL
                 into the 'Z' flag.

BIT bit,(IX+d)  :test specified bit in the byte pointed to by
BIT bit,(IY+d)   IX and IY plus 'd'. The carry bit is
                 unaffected.
```

13.06 SET AND RESET

Still dealing with bit operations, but not testing, are the SET and RESET instructions. SET can be used to force single bits in a specified byte to 1, while RESET forces the specified bit to 0. The general form for the SET and RESET instructions follows that for the BIT instructions as above.

```
SET bit,r           RESET bit,r
SET bit,(HL)        RESET bit,(HL)
SET bit,(IX+d)      RESET bit,(IX+d)
SET bit,(IY+d)      RESET bit,(IY+d)
```

13.Ø7 ROTATE AND SHIFT

Another group of bit instructions are the rotate and shift instructions. All of these instructions involve a kind of 'musical chairs' operation where bits are either bumped to the right or left of their current positions in the target byte.

RL r : RL (HL) : RL (IX+d) : RL (IY+d)

:the contents of the specified byte are shifted left (the direction from bit Ø to bit 7 is left) one bit position through the carry bit of the flag register. The carry bit is loaded into bit Ø and bit 7 is loaded into carry.

The single-byte RLA instruction is equivalent to the RL A version of the general two-byte RL r instruction given above.

RLC r : RLC (HL) : RLC (IX+d) : RLC (IY+d)

:the contents of the specified byte are rotated left one bit position. Bit 7 is loaded into bit Ø and also the carry.

Once again, there is a single-byte RLCA instruction equivalent to the RLC A version of the general two-byte RLC r instruction.

There are equivalent right rotate instructions (where bit Ø is loaded into the carry instead of bit 7):-

RR r : RR (HL) : RR (IX+d) : RR (IY+d) : RRA
RRC r : RRC (HL) : RRC (IX+d) : RRC (IY+d) : RRCA

Shift instructions are essentially the same as rotate instructions except the bit that is shifted out the end of the byte is not wrapped around to the other end.

There are two types of shifts; Arithmetic and Logical:-

SLA r : SLA (HL) : SLA (IX+d) : SLA (IY+d)

:Shift Left Arithmetic; bit 7 is loaded into the carry, all others are shifted left by one bit position, bit Ø is loaded with a 'Ø'.

SRA r : SRA (HL) : SRA (IX+d) : SRA (IY+d)

:Shift Right Arithmetic; bit Ø is loaded into the carry, all others are shifted right by one bit position, bit 7 is copied into bit 6 (but bit 7 remains unchanged).

SRL r : SRL (HL) : SRL (IX+d) : SRL (IY+d)

:Shift Right Logical; the same as Shift Right Arithmetic, but bit 7 is loaded into bit 6 and bit 7 is loaded with a Ø.

13.Ø8 INCREMENT AND DECREMENT

In a previous example dealing with incrementing a pointer through memory we used the HL register to contain the current pointer value. There are a group of instructions to not only increment, but also decrement, the contents of registers as well as specified bytes given by:-

INC r : INC rr : INC (HL) : INC (IX+d) : INC (IY+d)
DEC r : DEC rr : DEC (HL) : DEC (IX+d) : DEC (IY+d)

where 'rr' can be BC, DE, HL, IX, IY or SP.

13.Ø9 INPUT AND OUTPUT INSTRUCTIONS

Not only can we move data around in memory, in and out of registers, but also between the computer itself and external devices. That is, we can output 8-bit data onto, or read 8-bit data from, external data lines.

These sets of eight data lines are called collectively, 'PORTS'. An example of a port is the output connection to a printer which supplies the 8-bit data for characters of a hard copy listing. There are only two variations for each of the input and output port instructions:-

IN A,(n) :load the A register with the data connected to the input data lines of port number 'n'.

OUT (n),A :load the output data lines of port number 'n' with the data contained in the A register.

IN r,(C) :load register 'r' with the data connected to the input data lines of the port number specified by register C.

OUT (C),r :load the output data lines of the port number specified by C with the data contained in register 'r'.

13.10 STACK OPERATIONS

There are two classes of stack operations. One occurs as the result of direct access to the stack, while the other occurs indirectly because of 'CALL' and 'RET' instructions. Data must be 16-bit, and, in line with Z80 convention, the low order byte is placed in memory first. The SP register always points to the current low order byte. Remember, the stack grows **downwards** in memory. Firstly, the direct access instructions.

PUSH rr :the SP is decremented by one and the contents of the high order byte of 'rr' is loaded into memory. The SP is decremented by one again, the contents of the low order byte of 'rr' is loaded into memory. The register pair 'rr' includes AF, BC, DE, HL, IX and IY.

POP rr :the reverse operation to PUSH. The byte in memory pointed to by SP is loaded into the low order byte of the register pair 'rr', the SP register is incremented by one, and the byte now pointed to by SP is loaded into the high order byte of 'rr'. Lastly, SP is incremented by one to point to the low order byte of the next 16-bit value up in memory.

Indirect stack operation results from the use of the 'CALL' and 'RET' instructions. When a CALL instruction is executed, the CPU needs to store a return address so that it can continue execution at that point in the main program from which the subroutine was called.

The CPU uses the stack to store this 16-bit return address in the same manner in which the 16-bit contents of a register pair is stored when a PUSH instruction is executed. When a return is made to the main program, the 16-bit return address is loaded from the stack into the PC register.

That is, the execution of a RET instruction is similar in operation to the POP instruction which retrieves a 16-bit value from the stack and loads it into a register pair.

Note that even within a called subroutine there can be a CALL to another subroutine and so on. The stack then will grow downward with each new CALL, only being shrunk by a RET instruction.

13.11 OTHER INSTRUCTIONS

The above covers the most commonly used instructions and for details of the operation of the balance of the instructions I refer you to one of the many publications which describe all operations of the Z80 completely.

-ooOoo-

CHAPTER 14

INPUT AND OUTPUT - THE REAL WORLD OUTSIDE

There are two types of input/output (I/O) in the VZ200/300 - standard I/O and memory-mapped I/O. Cassette tape read/write operations are memory-mapped I/O, as is the scanning of the keyboard to detect keypresses. Standard I/O operations are used for printer output and handshaking as well as for disc controller operations.

14.01 STANDARD I/O

Standard I/O makes use of the special I/O instructions which can be performed by the Z-80 CPU. There are two groups of these instructions:-

(i) IN A,(n) ; OUT (n),A

(ii) IN r,(C) ; OUT (C),r

where 'n' is an 8-bit address and 'r' is an 8-bit register.

There is a third group of 'block move' I/O instructions which are essentially the same as the two groups above but handle blocks of data instead of single bytes.

The 'IN' instruction reads one byte from an external device into an 8-bit register, while the 'OUT' instruction transfers a data byte from an 8-bit register out to an external device.

The I/O instruction is special because it uses an I/O address which is an 8-bit address completely separate from the normal memory address map. The 8-bit address is contained in the I/O instruction itself ['n' in IN A,(n); OUT (n),A] or previously loaded into the C register [IN r,(C); OUT (C),r].

As the I/O address is specified by an 8-bit byte the I/O address can be from 0 to 255 allowing up to 256 external devices to be selected.

The I/O sequence is a slight modification of the normal memory read/write operation and uses the bottom eight address lines (A0-A7) to communicate the required I/O address to external devices. The devices are alerted that address lines A0-A7 are being used to output the I/O address by a separate IORQ output line.

After decoding the I/O address to see which device is to be selected, the selected device checks to see if the CPU requires an input or an output operation by reading the RD and WR output lines from the CPU and acts accordingly.

Although there are 256 I/O PORTS available which can be separately addressed, the VZ200/300 uses only a small portion of these. I/O ports are used for the printer interface, disc controller, joystick and the memory bank switch for the 64K memory expansion module.

The entire top half of the I/O address range is unused at present, that is, from I/O address 128 to 255 and so could be used to map devices other than the standard devices at present available for the VZ200/300.

We will now look in more detail at probably the most commonly used I/O port; the printer port.

14.02 THE PRINTER I/O PORT

Printers used with the VZ200/300 output port conform with a standard called the 'Centronics bus'. This standard defines the lines used for data transfer, handshaking, voltage levels and timing of the signals sent and received from the printer.

Handshaking control signals are used to synchronise the transfer of data from the CPU to the printer.

The first step in the normal sequence for the transfer of a byte of data (representing a character to be printed, a carriage return, line feed or part of a control sequence to set the printer to underline) is for the CPU to check to see if the printer is ready to receive the next character. This is done by reading the status on the BUSY/READY handshake from the printer.

If the CPU sends the character before the printer is ready to receive it, the character will be lost. When the printer is ready the character is loaded into the output latch mapped onto the printer I/O address. Finally, the CPU sends a pulse to the printer which strobes the data into the printer buffer. To send the next character the process is repeated.

Below are the port assignments for these operations:-

I/O OPERATION	I/O PORT ADDRESS
Read Printer Status	00H (0 decimal) [input]
Load Output Latch	0EH (14 decimal) [output]
Strobe Data to Printer	0DH (13 decimal) [output]

Bit Ø of the input byte from port ØØH is low (Ø) if the printer is ready and high (1) if the printer is busy.

One of the quirks of the printer driver routine is that it filters the data before it sends it out the port. This is no problem when just sending normal text to the printer during a LLIST operation, but if you are attempting to set up the printer by control codes from Basic using the LPRINT CHR$( ) command, the filter in the output routine prevents you from outputting certain codes.

To get around this restriction you can directly output the codes to the printer port using the Basic 'OUT p,n' command, where 'p' is the port address and 'n' is the data to be output. The following short listing will allow outputting a data byte from Ø-255 to the printer port including handshake.

Listing 14.1 Direct Output to the Printer Port.

```
1ØØ IF (INP(Ø)AND1)<>Ø THEN 11Ø: REM CHECK IF PRINTER READY
11Ø OUT 14,D                    : REM LOAD OUTPUT LATCH
12Ø OUT 13,D                    : REM STROBE DATA INTO PRINTER
```

This program is particularly useful if you want to send graphic codes to the printer as the filter in the printer driver stops some codes between Ø and 32 decimal from being sent to the printer when using the LPRINT CHR$() command.

The equivalent M/C code could be as follows in Listing 12.2.

Listing 12.2 Machine Code Output to the Printer Port

```
CHK   IN    A,(ØØH)
      BIT   Ø,A
      JR    NZ,CHK
      LD    A,data
      OUT   (ØEH),A
      OUT   (ØDH),A
```

14.Ø3 MEMORY-MAPPED I/O

The second type of I/O is memory-mapped I/O. The external device looks for a particular address on all of the address lines A0-A15. That is, each external device looks just like a memory location access for read and write.

As all sixteen address lines are being used it is possible to have 65536 separate external devices selected. The catch is that these addresses are part of the memory map for the ROMs and RAMs and not separate as is the case for standard I/O and so a careful decision as to what part of the memory range is going to be allocated to the I/O has to be made.

In the VZ200/300 the address space from 6800H-6FFFH has been allocated to memory mapped I/O for keyboard scanning, cassette input and output, sound and video attributes. This area has not been fully decoded so in the case of cassette I/O, sound and video attributes, a read or write to any address within the range 6800H-6FFFH will work. In the case of keyboard scanning the situation is a little more complex because row selection is driven from the state of the low-order address lines (see Appendix 4).

A Basic POKE to 6800H (26624 decimal) will load the data into the output latch decoded to this address range. The bits of this output latch are allocated as follows:-

BIT NUMBER	FUNCTION
0	One side of a push-pull drive for the output speaker (see bit 5).
1	Unused.
2	Cassette output drive line.
3	Mode [0=mode(0), 1=mode(1)]
4	Background colour.
5	Other side of push-pull drive to speaker.
6	Unused.
7	Unused.

Notice that in order to alter the contents of one bit you need to load in all eight bits of the latch. Therefore you need to keep track of the state of the other bits in the latch to prevent, say, the toggling of the cassette output bit from causing spurious output from the speaker or switching the mode of the video. For this reason a copy of the output latch is maintained at 783BH (30779 decimal) and should be updated with any new pattern that you are outputting to the latch.

Listing 12.3 is a Basic demonstration program which shows how to switch screen modes without using the MODE( ) command:-

Listing 12.3

```
100 L=PEEK(30779)           :REM LOOK UP COPY OF LATCH
110 M=(L) OR (8)            :REM MAKE BIT 3 = 1 (MODE(1))
120 POKE 26624,M            :REM UPDATE OUTPUT LATCH
130 POKE 30779,M            :REM UPDATE COPY OF LATCH
140 FOR D=1 TO 500:NEXT D:REM DELAY TO SEE WHATS HAPPENING
150 L=PEEK(30779)           :REM LOOK UP COPY OF LATCH
160 M=(L) AND (55)          :REM MAKE BIT 3 = 0 (MODE(0))
170 POKE 26624,M            :REM UPDATE OUTPUT LATCH
180 POKE 30779,M            :REM UPDATE COPY OF LATCH
190 FOR D=1 TO 500:NEXT D:REM DELAY TO SEE
200 GOTO 100                :REM DO IT AGAIN
```

14.Ø4 CASSETTE INPUT

Reading the cassette input line is done by PEEKing location 6800H and testing the level of bit 6. As the signal of the audio data comes in from the cassette recorder the level on bit 6 changes from one to zero. There are special software routines in the VZ200/300 ROM to decipher these ones and zeros into bytes which form the program to be loaded into memory.

Actually, it is a pretty useless exercise to use Basic to read this input line as any changes are occurring too fast for Basic to catch all the data. Machine code must be used in this area.

Connecting a computer to the outside real world is one of the more fascinating aspects of micro-computing and opens up another world of real-time data acquisition, digitizers, and control of external hardware devices. Venturers into this world require not only good knowledge in the software area but also expertise in hardware design.

-ooOoo-

NOTES

CHAPTER 15

CONCLUSION - WHERE TO GO FROM HERE

It is not possible to cover all aspects of the scope of machine code programming in this introductory text, indeed, it is not even possible to go deep enough into the areas covered. Where this has caused frustration I apologise, but at least it indicates a desire on your part to go on to a more deeper understanding of the subject.

This text was intended to provide you with the means to carry out simple programming exercises on paper and on your VZ200/300. At least that was the aim of this book.

The next step would be to try and modify programming examples from other Z-80 machines as now you should have a better understanding of the structure of your VZ200/300.

The next step I would like to undertake is to produce a handbook of machine code routines which could stand alone or be incorporated into larger programs, such utilities as tape header reading routines and the like.

Coverage in such detail shall have to be the subject of another text specifically aimed at covering specific programming examples, maybe in the near future.

Remember, you will not learn machine code from a book; you must learn from experience. You should try writing your own programs, or modifications of programs of other Z-80 machines.

GOOD LUCK !!!

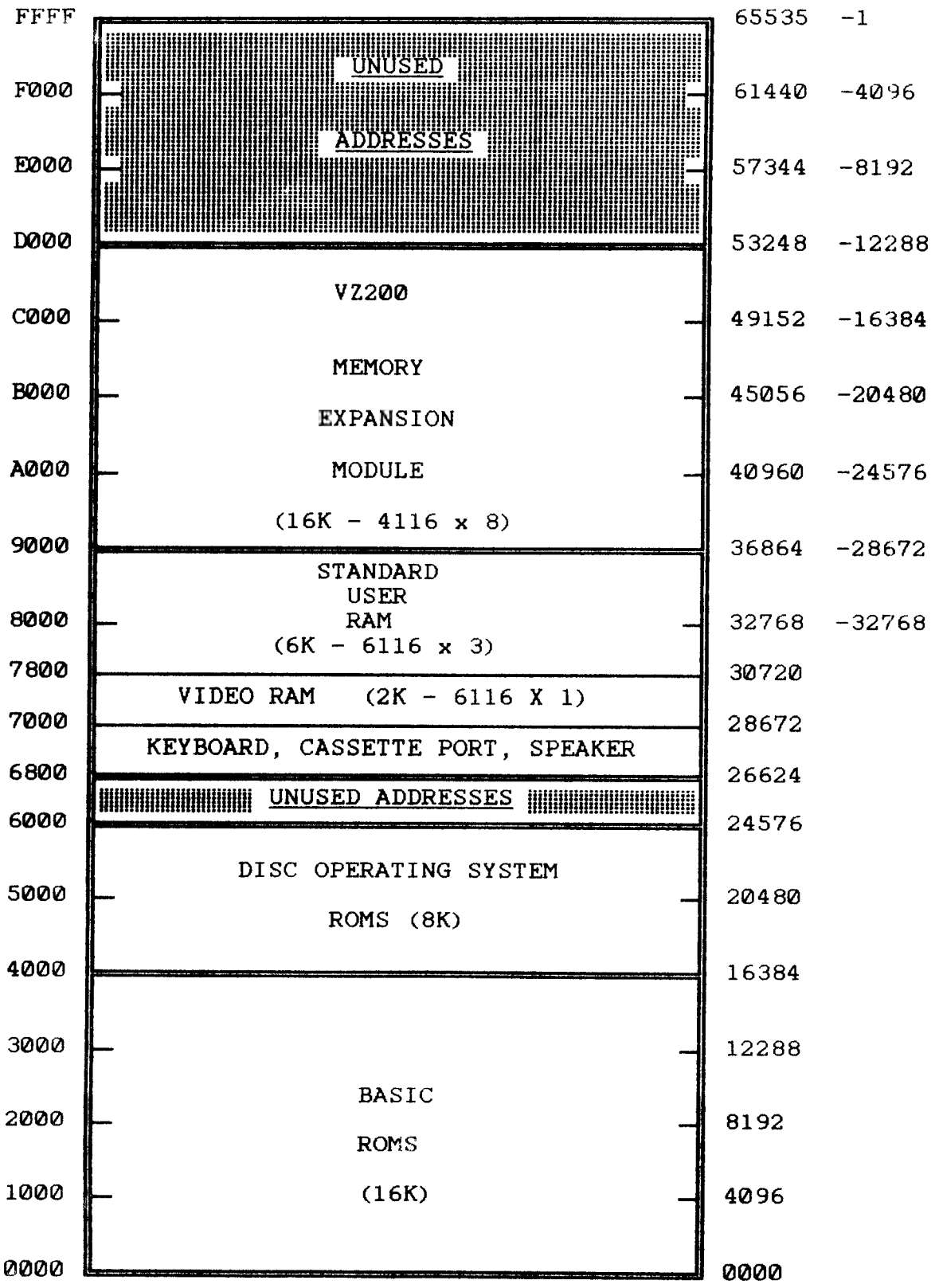
Steve Olney

---oooooOOOOOooooo---

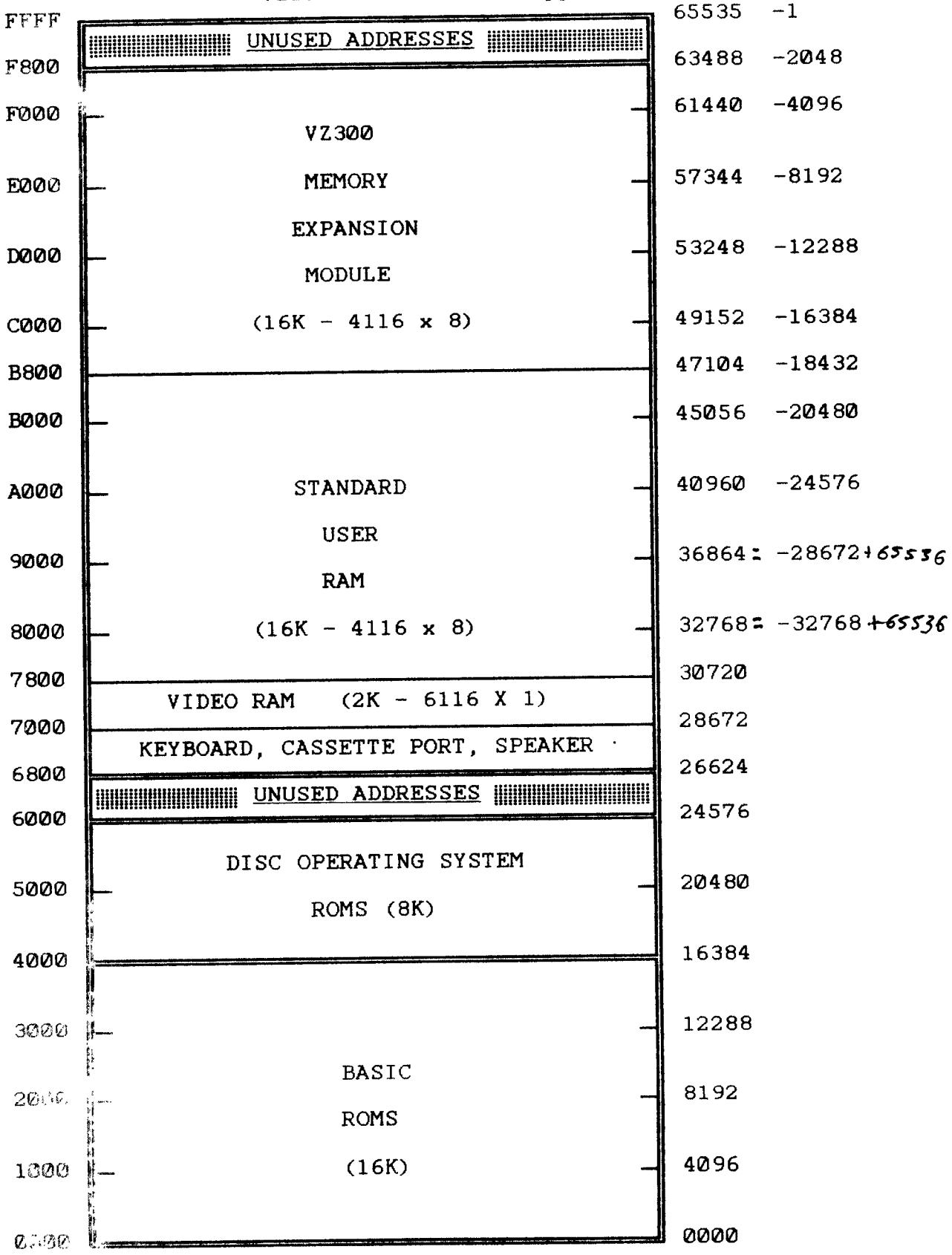
NOTES

APPENDIX 1

VZ200 MEMORY MAP

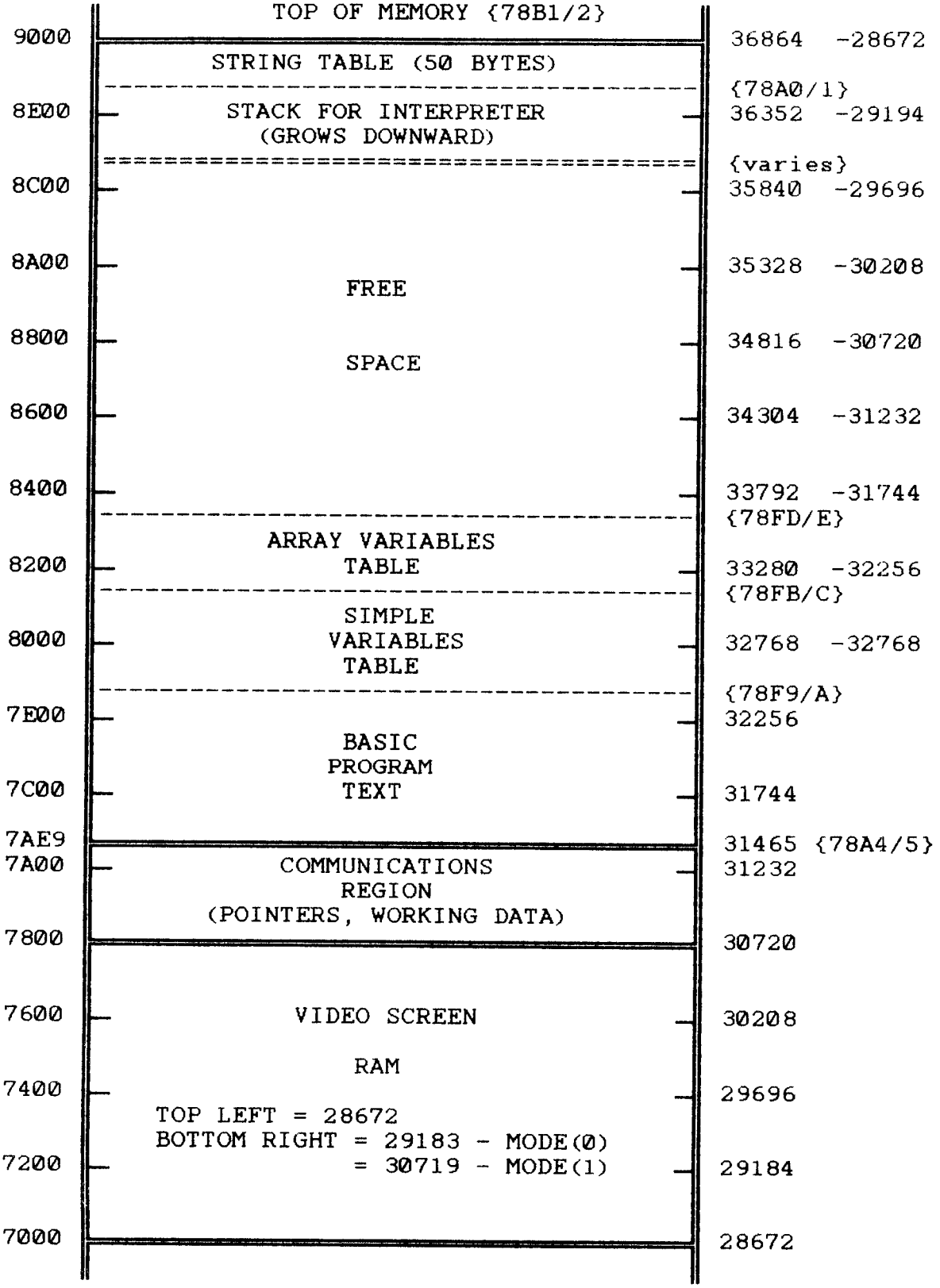


VZ300 MEMORY MAP - Appendix 1 cont'd



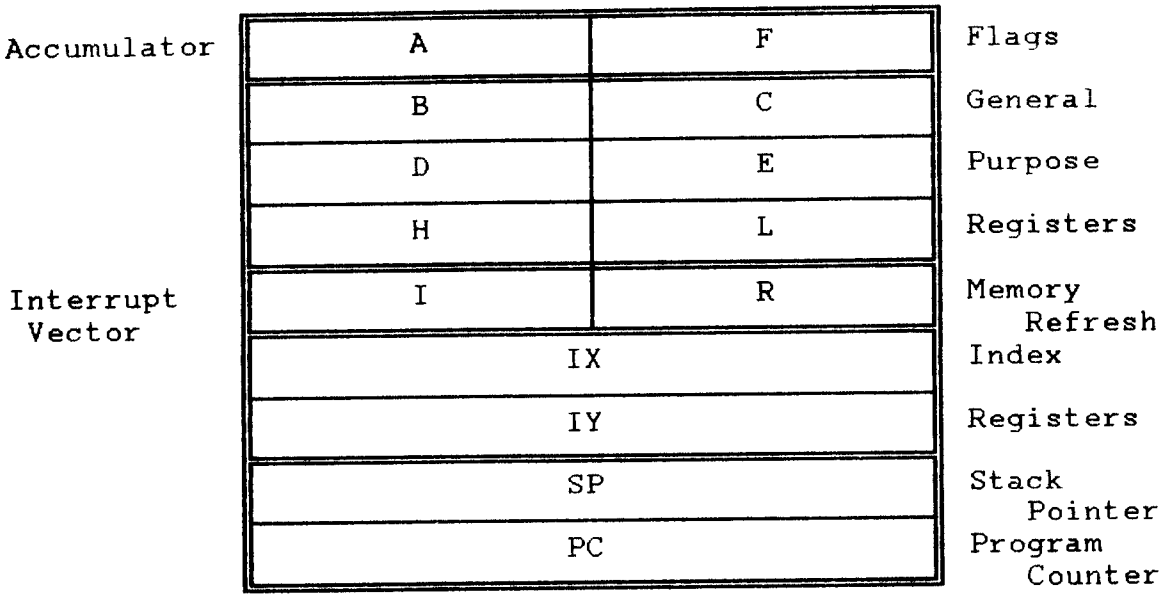
BASIC WORK SPACE FOR 8K VZ200 - Appendix 1 cont'd

Address values inside { } brackets give location of pointer inside communications area. (See Chapter 4 for details).

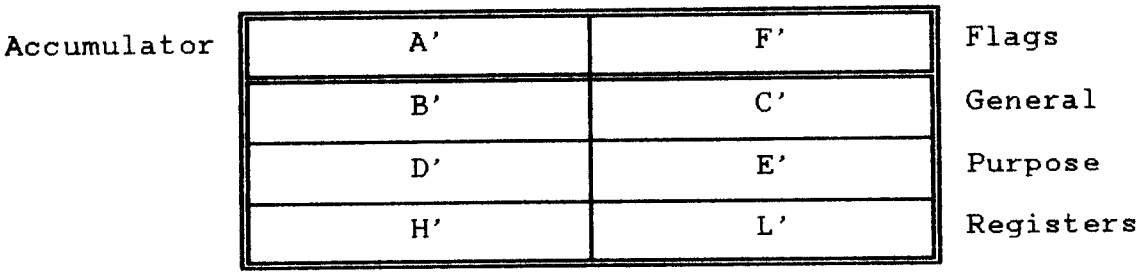


APPENDIX 2

Z80 REGISTERS

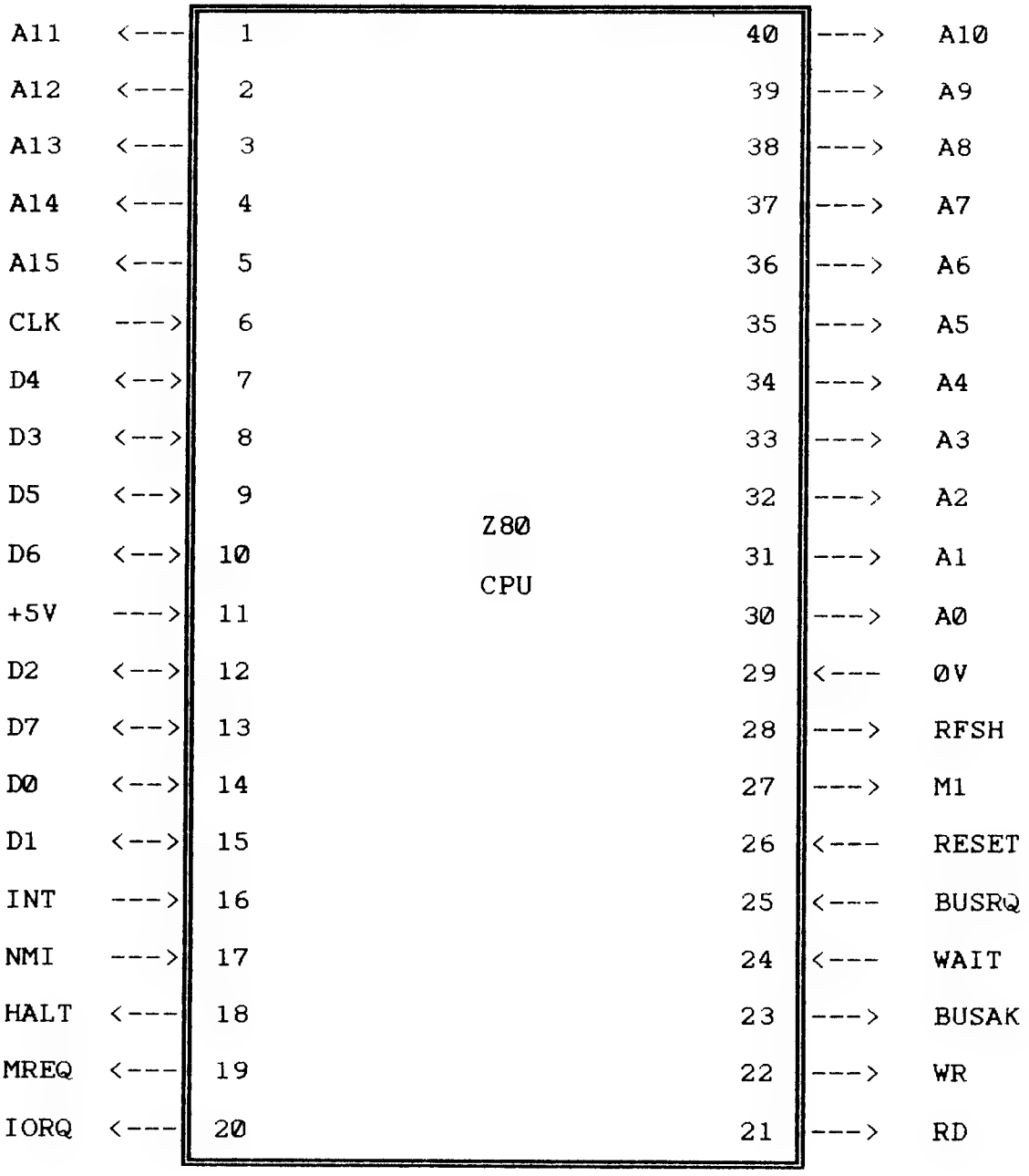


ALTERNATE REGISTER SET



APPENDIX 3

Z80 PINOUT



APPENDIX 4

KEYBOARD LAYOUT

The VZ200/300 keyboard matrix is memory-mapped I/O. One side of the key switches are wired to address lines A0-A7. The other side of the switches are wired to an input port which is read whenever any address in the range 6800H-6FFFH is selected.

That is, by PEEKing in this range, data can be read out of the input port as part of a normal memory PEEK. By PEEKing addresses in the range 6800H-6FFFH that correspond to only one of the address lines A0-A7 being low at any one time, we can select one of the horizontal ROWS in the table below.

By examining the value returned from the PEEK, we can determine which bit is low and hence in what COLUMN the depressed key lies. The intersection of the selected ROW and the detected COLUMN specifies which key was pressed.

By scanning through each ROW in turn and checking for one of the bits 0-5 for a low level, we can scan the whole keyboard. If we do not detect a low on any of the bits for all of the addresses, then no key has been depressed.

ADDR. LINE (LOW)	HEX ADDR.	BASIC PEEK()	BIT 0	BIT 1	BIT 2	BIT 3	BIT 4	BIT 5
A0	687FH	26751	H	L	:	K	;	J
A1	68BFH	26815	Y	O	RETURN	I	P	U
A2	68DFH	26847	6	9	-	8	0	7
A3	68EFH	26863	N	.	UNUSED	,	SPACE	M
A4	68F7H	26871	5	2	UNUSED	3	1	4
A5	68FBH	26875	B	X	SHIFT	C	Z	V
A6	68FDH	26877	G	S	CTRL	D	A	F
A7	68FEH	26878	T	W	UNUSED	E	Q	R

A quick check to see if ANY key has been depressed can be used to save time. By PEEKing address 6800H (26624 decimal) all address lines A0-A7 are low and a check to see if any bit 0-5 is low will show if any key is depressed. If a bit is low then the full scan would be done to find which one, while if no bits were found low then no keys were depressed.

The above refers to Basic PEEKing, but the method is easily implemented in M/C. Instead of PEEKing, use the M/C instruction: LD A,(68xx). Then check the state of each bit of the accumulator by rotating through the carry flag and testing.

-ooOoo-

APPENDIX 5

SYSTEM POINTER ADDRESSES

HEX	DECIMAL	FUNCTIONAL DESCRIPTION
783B	30779	Contains current state of the output latch.
787D/E/F	30845/6/7	Interrupt Exit. (fill with C3 YY XX where 'XX YY' is the address of your interrupt routine).
788E/F	30862/3	Contains the address of USER subroutine.
789C	30876	Output stream device code. 1=printer, 0=screen, -1=cassette.
78A0/1	30880/1	Address of bottom of string area.
78A2/3	30882/3	Current line number during execution.
78A4/5	30884/5	Address of start of program, (usually 7AE9H).
78A6	30886	Current cursor position across line, (0-31).
78AB	30891	Updated from Refresh Register during RND, (0-255).
78B1/2	30897/8	Top of memory pointer, (T.O.M.).
78DA/B	30938/9	Line number of last DATA value read during execution.
78E1	30945	Auto line numbering flag, 0=off, >0=on.
78E2/3	30946/7	Next line to be output in auto-line numbering.
78E4/5	30948/9	Auto line numbering increment.
78E6/7	30950/1	Address of start of current line in memory during execution.
78E8/9	30952/3	Current value of the SP at beginning of statement.
78F5/6	30965/6	Last line number executed when END or STOP.
78F9/A	30969/70	End of Basic pointer.
78FB/C	30971/72	Address of dimensioned variables table.
78FD/E	30973/4	Start address of free space.

Appendix 5 cont'd

HEX	DECIMAL	FUNCTIONAL DESCRIPTION
7901	30977	Start of variable type declaration list. There are 26 spaces, one for each letter of the alphabet. A declaration holds for all variables which start with the declared letter. Declarations are:- 2=integer, 3=string, 4=single precision, 8=double precision.
791A	31002	End of variable declaration list.
791B	31003	Trace flag (Ø=off, >Ø=on).
7921/2	31009/10	Value of 'X' passed in Y=USR(X).

-ooOoo-

DJNZ 10 OUT (n), A = D3
OR C B5
LD C ED 60
RST 38 DF

APPENDIX 6

COMMON Z-80 OPCODES

The following abbreviations have been used in this table:-

n = 8-bit number
r = 8-bit register
c = condition code
d = 8-bit offset or displacement
nn = 16-bit number
rr = 16-bit register pair
mm = 16-bit address

NOTE: - the 16-bit quantities 'nn' and 'mm' are two separate 8-bit bytes stored in memory with the low-order byte before the high-order byte.

ADC A,n - Add with Carry the 8-bit number 'n' to the accumulator.

ADC A,n	CE n

ADC A,r - Add with Carry the contents of the 8-bit register 'r' to the accumulator.

ADC A,A	8F
ADC A,B	88
ADC A,C	89
ADC A,D	8A
ADC A,E	8B
ADC A,H	8C
ADC A,L	8D

ADC HL,rr - Add with Carry the contents of register pair 'rr' to HL.

ADC HL,BC	ED4A
ADC HL,DE	ED5A
ADC HL,HL	ED6A

ADD A,n - Add without Carry the 8-bit number 'n' to the accumulator.

ADD A,n	C6 n

ADD A,r - Add without Carry the contents of the 8-bit register 'r' to the accumulator.

ADD A,A	87
ADD A,B	80
ADD A,C	81
ADD A,D	82
ADD A,E	83
ADD A,H	84
ADD A,L	85

Appendix 6 cont'd

ADD HL,rr - Add without Carry the contents of the 16-bit register pair 'rr' to HL.

ADD HL,BC	Ø9
ADD HL,DE	19
ADD HL,HL	29

CALL mm - Jump to a subroutine starting at address 'mm'.

CALL mm	CD mm

CALL c,mm - Jump to a subroutine starting at address 'mm' depending on the condition of the flag bits in the flag register.

CALL Z,mm	CC mm
CALL NZ,mm	C4 mm
CALL C,mm	DC mm
CALL NC,mm	D4 mm
CALL PE,mm	EC mm
CALL PO,mm	E4 mm
CALL M,mm	FC mm
CALL P,mm	F4 mm

CCF - Complement Carry flag.

CCF	3F

CP n - Compare the contents of the accumulator with the 8-bit number 'n'. Set the condition flags accordingly.

CP n	FE n

CP r - Compare the contents of the accumulator with the contents of the 8-bit register 'r'. Set the condition flags accordingly.

CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD

CP (HL) - Compare the contents of the accumulator with the contents of the memory location pointed to by HL.

CP (HL)	BE

Appendix 6 cont'd

DEC r - Decrement the 8-bit register 'r'by one.

DEC A	3D
DEC B	Ø5
DEC C	ØD
DEC D	15
DEC E	1D
DEC H	25
DEC L	2D

DEC rr - Decrement the 16-bit register pair 'rr' by one.

DEC BC	ØB
DEC DE	1B
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B

DEC (HL) - Decrement the contents of the memory location pointed to by HL by one.

DEC (HL)	35

INC r - Increment the 8-bit register 'r' by one.

INC A	3C
INC B	Ø4
INC C	ØC
INC D	14
INC E	1C
INC H	24
INC L	2C

INC rr - Increment the 16-bit register pair 'rr' by one.

INC BC	Ø3
INC DE	13
INC HL	23

INC (HL) - Increment the contents of the memory location pointed to by HL by one.

INC (HL)	34

JP mm - Jump to address 'mm'.

JP mm	C3 mm

Appendix 6 cont'd

JP (rr) - Jump to the address 'mm' held in register pair 'rr'.

JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9

JP c,mm - Jump to the address 'mm' depending on the condition of the flag bits in the flag register.

JP Z,mm	CA mm
JP NZ,mm	C2 mm
JP C,mm	DA mm
JP NC,mm	D2 mm
JP PE,mm	EA mm
JP PO,mm	E2 mm
JP M,mm	FA mm
JP P,mm	F2 mm

JR d - Jump back or forwards in memory by the displacement 'd'.

JR d	18 d

JR c,d - Jump by the displacement 'd' depending on the condition of the flag bits in the flag register.

JR NZ,d	20 d
JR Z,d	28 d
JR NC,d	30 d
JR C,d	38 d

LD r,n - Load the 8-bit register 'r' with the 8-bit number 'n'.

LD A,n	3E n
LD B,n	06 n
LD C,n	0E n
LD D,n	16 n
LD E,n	1E n
LD H,n	26 n
LD L,n	2E n

LD rr,nn - Load the 16-bit register pair 'rr' with the 16-bit number 'nn'.

LD BC,nn	01 nn
LD DE,nn	11 nn
LD HL,nn	21 nn
LD IX,nn	DD21 nn
LD IY,nn	FD21 nn
LD SP,nn	31 nn

Appendix 6 cont'd

LD A,(mm) - Load the accumulator with the contents of memory location 'mm'.

LD A,(mm)	3A mm

LD rr,(mm) - Load the register pair 'rr' with the contents of memory locations 'mm' and 'mm+1'.

LD BC,(mm)	ED4B mm
LD DE,(mm)	ED5B mm
LD HL,(mm)	2A mm

LD A,r - Load the accumulator with contents of the 8-bit register 'r'.

LD A,A	7F
LD A,B	78
LD A,C	79
LD A,D	7A
LD A,E	7B
LD A,H	7C
LD A,L	7D

LD r,(rr) - Load the 8-bit register 'r' with the contents of the memory location pointed to by the 16-bit register pair 'rr'.

LD A,(BC)	ØA
LD A,(DE)	1A
LD A,(HL)	7E
LD B,(HL)	46
LD C,(HL)	4E
LD D,(HL)	56
LD E,(HL)	5E
LD H,(HL)	66
LD L,(HL)	6E

LD (mm),A - Load the contents of the accumulator into memory location 'mm'.

LD (mm),A	32 mm

LD (mm),rr - Load the contents of the 16-bit register pair 'rr' into memory locations 'mm' and 'mm+1'.

LD (mm),BC	ED43 mm
LD (mm),DE	ED53 mm
LD (mm),HL	22 mm

Appendix 6 cont'd

LD (rr),r - Load the memory location pointed to by the 16-bit register pair 'rr' with contents of the 8-bit register 'r'.

LD (BC),A	Ø2
LD (DE),A	12
LD (HL),A	77
LD (HL),B	7Ø
LD (HL),C	71
LD (HL),D	72
LD (HL),E	73
LD (HL),H	74
LD (HL),L	75

LD (rr),n - Load the memory location pointed to by the 16-bit register pair 'rr' with the 8-bit number 'n'.

LD (HL),n	36

RET - Return from a call to a subroutine.

RET	C9

RET c - Return from a call to a subroutine depending on the condition of the flag bits in the flag register.

RET Z	C8
RET NZ	CØ
RET C	D8
RET NC	DØ
RET PE	E8
RET PO	EØ
RET M	F8
RET P	FØ

SBC A,n - Subtract with Carry the 8-bit number 'n' from the contents of the accumulator. Store the result back in the accumulator.

SBC A,n	DE n

SBC A,r - Subtract with Carry the contents of the 8-bit register 'r' from the contents of the A register. Store the result back in the A register.

SBC A,A	9F
SBC A,B	98
SBC A,C	99
SBC A,D	9A
SBC A,E	9B
SBC A,H	9C
SBC A,L	9D

Appendix 6 cont'd

SBC HL,rr - Subtract the contents of the 16-bit register pair 'rr' from HL. Store the result back in HL.

SBC HL,BC	ED42
SBC HL,DE	ED52
SBC HL,HL	ED62

SBC A,(HL) - Subtract with Carry the contents of the memory location pointed to by HL from the contents of the accumulator. Store the result back in the accumulator.

SBC A,(HL)	9E

SCF - Set the Carry flag bit in the flag register.

SCF	37

SUB n - Subtract the 8-bit number 'n' from the contents of the accumulator. Store the results back in the accumulator.

SUB n	D6 n

SUB r - Subtract the contents of the 8-bit register 'r' from the contents of the accumulator. Store the result back in the accumulator.

SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95

SUB (HL) - Subtract the contents of the memory location pointed to by HL from the accumulator. Store the result back in the accumulator.

SUB (HL)	96

-ooOoo-

APPENDIX 7

HEXADECIMAL/DECIMAL CONVERSION TABLES

Positive Numbers FIRST HEX DIGIT

SECOND HEX DIGIT	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Negative Numbers FIRST HEX DIGIT

SECOND HEX DIGIT	F	E	D	C	B	A	9	8
F	-1	-17	-33	-49	-65	-81	-97	-113
E	-2	-18	-34	-50	-66	-82	-98	-114
D	-3	-19	-35	-51	-67	-83	-99	-115
C	-4	-20	-36	-52	-68	-84	-100	-116
B	-5	-21	-37	-53	-69	-85	-101	-117
A	-6	-22	-38	-54	-70	-86	-102	-118
9	-7	-23	-39	-55	-71	-87	-103	-119
8	-8	-24	-40	-56	-72	-88	-104	-120
7	-9	-25	-41	-57	-73	-89	-105	-121
6	-10	-26	-42	-58	-74	-90	-106	-122
5	-11	-27	-43	-59	-75	-91	-107	-123
4	-12	-28	-44	-60	-76	-92	-108	-124
3	-13	-29	-45	-61	-77	-93	-109	-125
2	-14	-30	-46	-62	-78	-94	-110	-126
1	-15	-31	-47	-63	-79	-95	-111	-127
0	-16	-32	-48	-64	-80	-96	-112	-128

NOTE: Another method of finding two's complement representation of an 8-bit negative number is to add 256 to the negative number and then look up the resulting positive quantity in the top table. e.g. to find the two's complement of -10, 256-10 = 246 = 0F6H.

APPENDIX 8

Z-80 MNEMONICS RECOGNISED BY THE VZ EDITOR ASSEMBLER

ADC HL,ss
ADD A,r
ADD A,(IY+d)
ADD IY,rr
BIT b,(IX+d)
CALL cc,nn
CP s
CPI
DAA
DEC IY
DJNZ e
EX (SP),IX
EX DE,HL
IM 0
IN A,(n)
INC IX
INC (IY+d)
IND
INIR
JP (IY)
JR C,e
JR NZ,e
LD A,(DE)
LD A,R
LD (HL),A
LD HL,(nn)
LD IX,nn
LD (IX+d),r
LD (IY+d),n
LD (nn),dd
LD (nn),IY
LD r,(IX+d)
LD r,r'
LD SP,IY
LDI
NOP
OTIR
OUTD
POP IY
PUSH IY
RET
RETN
RLC (HL)
RLC r
RR m
RRCA
SBC A,s
SET b,(HL)
SET b,r
SRL m

ADC A,s
ADD A,(HL)
ADD HL,ss
AND s
BIT b,(IY+d)
CALL nn
CPD
CPIR
DEC m
DEC ss
EI
EX (SP),IY
EXX
IM 1
IN r,(C)
INC (IX+d)
INC r
INDR
JP (HL)
JP cc,nn
JR e
JR Z,e
LD A,I
LD (BC),A
LD dd,nn
LD (HL),r
LD IX,(nn)
LD IY,nn
LD (IY+d),r
LD (nn),HL
LD R,A
LD r,(IY+d)
LD SP,HL
LDD
LDIR
OR s
OUT (C),r
OUTI
POP qq
PUSH qq
RET cc
RL m
RLC (IX+d)
RLCA
RRA
RRD
SBC HL,ss
SET b,(IX+d)
SLA m
SUB s

ADD A,n
ADD A,(IX+d)
ADD IX,pp
BIT b,(HL)
BIT b,r
CCF
CPDR
CPL
DEC IX
DI
EX (SP),HL
EX AF,AF'
HALT
IM 2
INC (HL)
INC IY
INC ss
INI
JP (IX)
JP nn
JR NC,e
LD A,(BC)
LD A,(nn)
LD (DE),A
LD dd,(nn)
LD I,A
LD (IX+d),n
LD IY,(nn)
LD (nn),A
LD (nn),IX
LD r,(HL)
LD r,n
LD SP,IX
LDDR
NEG
OTDR
OUT (n),A
POP IX
PUSH IX
RES b,m
RETI
RLA
RLC (IY+d)
RLD
RRC m
RST p
SCF
SET b,(IY+d)
SRA m
XOR s

Appendix 8 cont'd

PSEUDO-OPS RECOGNISED BY THE VZ EDITOR ASSEMBLER

EQU nn EQU $ DEFB n DEFS nn DEFW nn

OPERAND NOTATION

r	one of the following registers: A, B, C, D, E, H, L.
( )	brackets around an operand point to the contents of a memory location or I/O port, pointed to by the operand. The operand could be an eight or sixteen bit value, the contents of an eight bit register or sixteen bit register pair, or the contents of an index register plus an eight bit displacement value.
n	an eight bit value in the range 0 to 255.
nn	a sixteen bit value in the range 0 to 65535.
d	an eight bit signed displacement value in the range -128 to 127.
b	bit number in the range of 0 to 7 (Bit 0 to Bit 7).
cc	state of the condition bits in the flag register: NZ, Z, NC, C, PO, PE, P, M.
e	an eight bit offset in the range -128 to 127.
qq	one of the following register pairs: AF, BC, DE, HL.
ss	one of the following register pairs: BC, DE, HL, SP.
pp	one of the following register pairs: BC, DE, IX, SP.
rr	one of the following register pairs: BC, DE, IY, SP.
s	one of the following: r, n, (HL), (IX+d), (IY+d).
dd	one of the following register pairs: BC, DE, HL, SP.
m	one of the following: r, (HL), (IX+d), (IY+d).

-ooOoo-